Université de Toulouse

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :
Institut National Polytechnique de Toulouse (INP Toulouse)

Discipline ou spécialité :

Réseaux, Télécommunications, Systèmes et Architecture

Présentée et soutenue par :
Thomas Mégel

le : mardi 3 avril 2012

Titre :

Placement, ordonnancement et mécanismes de migration de tâches temps-réel pour des architectures distribuées multicœurs

---

Ecole doctorale :
Mathématiques Informatique Télécommunications (MITT)

Unité de recherche :
CEA LIST - IRIT

Directeur(s) de Thèse :

Christian Fraboul, Professeur INP/ENSEEIHT Toulouse - IRIT
Vincent David, Ingénieur de recherche CEA LIST

Rapporteurs :

Isabelle Puaut, Professeur de l'Université de Rennes - IRISA
Yvon Trinquet, Professeur de l'IUT de Nantes - IRCCyN

Membre(s) du jury :

Emmanuel Grolleau, Professeur ENSMA - LISI
Mathieu Jan, Ingénieur de recherche CEA LIST
Pascal Sainrat, Professeur UPS Toulouse - IRIT

Résumé

Les systèmes temps-réel embarqués critiques intègrent un nombre croissant de fonctionnalités comme le montrent les domaines de l'automobile ou de l'aéronautique. Ces systèmes doivent offrir un niveau maximal de sûreté de fonctionnement en disposant des mécanismes pour traiter les défaillances éventuelles et doivent être également performants, avec le respect de contraintes temps-réel strictes. Ces systèmes sont en outre contraints par leur nature embarquée : les ressources sont limitées, tels que par exemple leur espace mémoire et leur capacité de calcul. Dans cette thèse, nous traitons deux problématiques principales de ce type de systèmes.

La première porte sur la manière d'apporter une meilleure tolérance aux fautes dans les systèmes temps-réel distribués subissant des défaillances matérielles multiples et permanentes. Ces systèmes sont souvent conçus avec une allocation statique des tâches. Une approche plus flexible effectuant des reconfigurations est utile si elle permet d'optimiser l'allocation à chaque défaillance rencontrée, pour les ressources restantes. Nous proposons une telle approche hors-ligne assurant un dimensionnement adapté pour prendre en compte les ressources nécessaires à l'exécution de ces actions. Ces reconfigurations peuvent demander une réallocation des tâches ou répliques si l'espace mémoire local est limité. Dans un contexte temps-réel strict, nous définissons notamment des mécanismes et des techniques de migration garantissant l'ordonnançabilité globale du système.

La deuxième problématique se focalise sur l'optimisation de l'exécution des tâches au niveau local dans un contexte multicœurs préemptif. Nous proposons une méthode d'ordonnancement optimal disposant d'une meilleure extensibilité que les approches existantes en minimisant les surcoûts : le nombre de changements de contexte (préemptions et migrations locales) et la complexité de l'ordonnanceur.

Summary

Critical real-time embedded systems are integrating an increasing number of functionalities, as shown in automotive domain or aeronautics. These systems require high dependability including mechanisms to handle possible failures and have to be effective, meeting hard real-time constraints. These systems are also constrained by their embedded nature : resources are limited, such as their memory and their computing capacities. In this thesis, we focus on two main problems for this type of systems.

The first one is about a way to bring a better fault-tolerance in distributed real-time systems when multiple and permanent hardware failures can occur. In classical systems, the design is limited to a static task assignment. A more flexible approach exploiting reconfigurations is useful if it allows to optimize assignment at each failure for the remaining resources. We propose an offline approach to obtain an adapted sizing taking into account necessary resources to execute these actions. These reconfigurations may require to reallocate tasks or replicas if memory capacities are limited. In a hard real-time context, we define mechanisms and migration techniques to guarantee global schedulability of the system.

The second problem focus on optimizing performance to run tasks at a local level in a multicore preemptive context. We propose an optimal scheduling method allowing a better scalability than existing approaches by minimizing overheads : the number of context switches (local preemptions and migrations) and the scheduler complexity.

Remerciements

Mes remerciements vont d'une manière générale à toutes les personnes qui ont contribué à la réussite de ce travail au long de ces trois années.

Je remercie tout d'abord Vincent David et Christian Fraboul pour leur encadrement et pour la confiance qu'ils m'ont accordée pendant la réalisation de ces travaux de thèse.

Je tiens également à remercier les rapporteurs Isabelle Puaut et Yvon Trinquet pour leur relecture minutieuse, leur implication dans l'évaluation de ce mémoire.

Je tiens à exprimer ma gratitude envers Emmanuel Grolleau, Pascal Sainrat et Mathieu Jan qui m'ont fait l'honneur de leur présence dans le jury de thèse.

À toutes les personnes que j'ai pu côtoyer au laboratoire (LaSTRE), qui ont contribué par leur accueil, leurs discussions scientifiques et leurs conseils à la bonne ambiance générale ; je pense également aux bons moments passés à la pause café et au resto 1.

Un grand merci à tous ceux qui ont relu mes articles lors de leur rédaction de même que les différentes parties de ce mémoire. Pour leur expertise technique et leurs conseils avisés, je tiens à remercier Renaud Sirdey, Mathieu Jan et Matthieu Lemerre.

Je remercie tous les anciens collègues et amis qui m'ont permis de me changer les idées tout au long de cette période, ils se reconnaîtront.

Merci enfin à mes parents et ma famille pour leur soutien.

Table des matières

Chapitre 1

Introduction

Sommaire

1.1 Présentation du contexte général

La conception de systèmes temps-réel embarqués se développe de plus en plus avec l'intégration croissante de fonctionnalités critiques dans les processus industriels par exemple dans les domaines de l'automobile ou de l'aéronautique. Dans l'automobile, on voit l'apparition des technologies *X-by-wire* qui ont pour rôle de remplacer les systèmes de contrôle traditionnellement construits mécaniquement (frein, conduite, etc.). Dans l'aéronautique, c'est l'avionique modulaire intégrée (IMA) qui est la tendance de conception récente afin de remplacer des calculateurs dédiés par des calculateurs généralistes qui factorisent des fonctionnalités différentes, apportant une modularité de conception. Ce type de système embarqué est complexe à concevoir et à maîtriser. Cette complexité est de nos jours plus importante et ce pour plusieurs raisons.

La tendance actuelle est à l'intégration croissante de fonctionnalités critiques autrefois réalisées mécaniquement et maintenant développées au niveau logiciel : on parle de tâches critiques lorsque leur dysfonctionnement peut entraîner des conséquences graves.

Le système doit répondre à des exigences importantes souvent antinomiques : il faut construire des systèmes sûrs de fonctionnement ayant des mécanismes pour traiter les défaillances, mais le système doit rester performant car contraint par sa nature embarquée (ressources limitées, consommation d'énergie limitée...). Certains de ces systèmes doivent de plus répondre à des contraintes temps-réel strictes c'est à dire que non seulement les traitements doivent être corrects mais aussi réalisés dans le temps imparti.

Le besoin croissant en capacité de calcul et la maîtrise des coûts de production amènent de plus en plus à des architectures distribuées (réparties) et à l'utilisation de composants sur

étagère (*Commercial Off-The-Shell*). Ces composants sont moins chers, produits en grande quantité mais pas conçus dans l'optique de satisfaire les exigences spécifiques en terme de sûreté de fonctionnement. Ils présentent donc une fiabilité inférieure aux composants dédiés (sur mesure).

Nous nous intéressons donc à l'étude, la conception et la validation de systèmes embarqués temps-réel distribués (ou répartis) pour des applications critiques. Notre objectif est de rendre plus **tolérants** et plus **performants** les systèmes distribués temps-réel critiques.

La tolérance aux fautes consiste à fournir un service conforme aux spécifications en présence de fautes par redondance. Ces fautes peuvent être de nature externe (liées à l'environnement) ou interne (e.g. faute de conception). La redondance peut être matérielle, on emploie alors des composants identiques qui ont la même fonction dans le système. La réplication logicielle est la plus répandue, elle consiste à utiliser des copies des tâches, chacune étant exécutée sur des calculateurs différents, afin de maintenir des copies de l'état d'exécution de chaque tâche. Cette capacité du système à tolérer les défaillances est souvent limitée par une conception où les tâches et leur répliques sont statiquement allouées. Toutefois il y a un intérêt à autoriser une certaine flexibilité dans l'exécution d'un système distribué. Cette flexibilité opérationnelle est bénéfique lorsque des défaillances se produisent, entraînant la perte de ressources et des tâches allouées. Lors d'une défaillance, il est intéressant d'optimiser le placement des tâches par exemple pour améliorer le niveau de tolérance aux fautes ou pour l'équilibre de la charge sur les ressources restantes. Nous proposons de mettre en place une approche hors-ligne permettant cette flexibilité en-ligne de manière efficace et sûre pour l'allocation des tâches et répliques s'exécutant sur un système subissant des défaillances multiples. Nous nous intéresserons au niveau global de l'architecture distribuée à la manière dont le système opère en gardant un comportement déterministe.

Parallèlement aux aspects flexibilité, nous nous intéressons à l'utilisation locale de processeurs multicœurs au sein d'un nœud : l'apparition récente des systèmes multicœurs vient augmenter le parallélisme et les capacités de calcul des systèmes. Ce type d'architecture amène à de nouvelles problématiques. L'une d'entre elle est la mise en place d'un ordonnancement efficace tout en maintenant des garanties théoriques fortes (optimalité temps-réel). Un des surcoûts importants dans les politiques optimales d'ordonnancement est le changement de contexte dû à la préemption ou à la migration d'une tâche d'un cœur à l'autre. Nous cherchons des ordonnancements temps-réel optimaux, sur des processeurs identiques, ayant pour objectif la minimisation des préemptions et des migrations de tâches. Enfin, il sera intéressant d'obtenir une complexité en-ligne faible pour l'exécution des tâches afin d'avoir une approche extensible.

1.2 Contributions principales

Nous apportons deux contributions principales concernant les systèmes embarqués temps-réel critiques. Ces contributions visent à répondre aux deux questions suivantes, l'une portant sur la tolérance aux fautes et l'autre sur l'efficacité lors de l'exécution. La première question qui se pose est la manière d'apporter une meilleure tolérance aux fautes dans les systèmes temps-réel distribués subissant des défaillances multiples et permanentes.

Comme indiqué précédemment, ces systèmes sont souvent conçus pour une allocation statique des tâches et de leur réplique obligeant le concepteur à trouver un compromis entre l'allocation initiale et le nombre de défaillances à tolérer. Nous proposons une méthode permettant d'optimiser l'allocation pour chaque défaillance rencontrée. Notre objectif est d'atteindre à chaque défaillance une allocation dite de référence optimisée pour les ressources restantes. Nous verrons

que ces reconfigurations doivent être effectuées en temps borné et de manière sûre.

Nous identifions les mécanismes nécessaires. Nous modélisons par formulation linéaire afin de spécifier correctement les allocations en présence de ces mécanismes dans un cadre général, i.e. dimensionner en prenant compte les ressources nécessaires pour l'exécution de la reconfiguration. Nous montrons également comment explorer tous les scénarios possibles de défaillances.

Ces transitions peuvent demander une réallocation des tâches ou répliques, c'est à dire de migrer l'espace mémoire de tâches d'un nœud à l'autre de par la limitation de l'espace mémoire de chaque nœud. Des techniques de migrations appropriées à notre contexte sont alors nécessaires. Nous montrons comment ce mécanisme de migration peut opérer de manière sûre dans un système temps-réel strict. Les techniques proposées permettent notamment des délais bornés et ne présentent pas de temps de gel, c'est à dire n'impliquant pas de pause dans l'exécution des tâches. Autoriser un temps de gel pourrait conduire à un système moins prédictible car introduisant de la variabilité dans le temps effectif d'exécution.

D'un point de vue pratique, nous mesurons par simulation l'impact des migrations sur l'exécution. Nos résultats montrent que le surcoût engendré reste relativement faible par rapport à la fréquence de migration des tâches et par rapport à la taille de l'espace mémoire envisageable dans un système embarqué.

La deuxième question à l'étude porte sur l'efficacité de l'exécution et du fonctionnement des tâches temps-réel strictes du système au niveau local en présence de multicœurs.

Nous proposons une méthode d'ordonnancement disposant d'une meilleure extensibilité en minimisant les surcoûts : minimisant les changements de contexte et en réduisant la complexité en-ligne. Contrairement aux approches antérieures, notre méthode est basée sur une construction hybride (hors-ligne et en-ligne) : l'une hors-ligne pour placer les travaux sur des intervalles successifs, l'autre en-ligne pour les exécuter dynamiquement.

Nous proposons également un modèle linéaire en nombres entiers afin d'exprimer la présence et la préemption d'un travail entre plusieurs intervalles. Pour compléter le modèle, nous proposons quatre fonctions économiques qui mènent à des résultats d'ordonnancement corrects mais différents en termes d'optimisation. La partie en-ligne consiste à exécuter des travaux avec un ordonnanceur approprié. Nous en proposons un de faible complexité, dynamique et générant peu de préemptions et de migrations de tâches.

Notre approche permet de réduire significativement le nombre de préemptions et migrations de travaux pour le modèle de tâches périodiques. Les résultats montrent que nous générons moins de préemptions et migrations que d'autres ordonnanceurs optimaux. Cependant, il doit être souligné que notre approche peut nécessiter un investissement significatif (plusieurs minutes) mais acceptable dans sa partie hors-ligne.

Nous examinons enfin la complexité de calcul de notre programme linéaire, celui-ci est NP-difficile au sens fort. D'un point de vue pratique, nous étudions aussi l'empreinte mémoire des données de travail de l'ordonnanceur (les ensembles de poids) qui s'avère être de l'ordre de quelques kbits pour des instances réalistes.

1.3 Plan du mémoire

La suite de ce mémoire est structurée en quatre grands chapitres.

Dans le chapitre 2, nous établissons tout d'abord plusieurs constats sur la nature des composants utilisés et sur le parallélisme. Nous décrivons les différents besoins en termes de performance, de sûreté de fonctionnement, d'utilisation des ressources et de flexibilité. Nous identifions des problèmes fondamentaux liés à la conception de systèmes embarqués temps-réel répartis et proposons un état de l'art général des solutions existantes. Nous présentons ensuite les problèmes adressés dans ce mémoire, c'est à dire correspondant aux deux principales contributions énoncées à la section précédente. Enfin, nous précisons les modèles de travail.

Le chapitre 3 décrit une méthode pour la réallocation de tâches lors de défaillances matérielles : nous parlons alors de reconfiguration dynamique. Cette méthode a pour but de répondre à la problématique d'apporter une meilleure tolérance aux fautes en permettant une flexibilité opérationnelle dans l'exécution du système. Nous montrons comment spécifier des allocations et comment modéliser les ressources nécessaires aux mécanismes de la reconfiguration. Nous décrivons les différentes étapes pour appliquer notre méthode, et l'illustrons à travers un exemple. La méthode décrite ici est générale, néanmoins nous voulons l'utiliser dans le contexte applicatif du temps-réel strict et cela nécessite des techniques de migration adaptées.

Le chapitre 4 vient compléter la méthode générale décrite au chapitre précédent en proposant le mécanisme principal manquant à l'application des reconfigurations en-ligne. Nous décrivons plusieurs politiques de migrations pour le temps-réel strict. Nous présentons une approche permettant de vérifier la faisabilité en fonction des caractéristiques de la tâche et de celles du réseau. La faisabilité de la migration ne garantit pas pour autant l'ordonnançabilité de l'ensemble des tâches du système en présence de migration. Pour garantir l'ordonnançabilité, nous exprimons les caractéristiques temporelles associées aux différentes techniques. Ceci nous permet d'être indépendant de la politique d'ordonnancement utilisé. Nous montrons comment intégrer ces migrations dans des tests d'ordonnançabilité. Par expérimentation nous montrons que les techniques garanties de migrations engendrent un surcoût limité à l'exécution du système.

Le chapitre 5 propose la deuxième contribution principale qui est spécifique à l'exécution des tâches sur multicœurs, au niveau local d'un système distribué i.e. un nœud intégrant un *MPSoC*. Il s'agit rendre performants l'exécution et l'ordonnancement des tâches temps-réel strictes. Nous montrons une approche pour produire des ordonnancements optimaux minimisant les préemptions locales, et par conséquent les migrations locales. Nous proposons une formulation appropriée par programmation linéaire et proposons un ordonnanceur local optimal, dynamique et de complexité faible. Les résultats obtenus par simulation montrent la compétitivité de l'approche par rapport à l'état de l'art existant.

Enfin nous concluons sur l'ensemble de cette étude au chapitre 6.

Liste des publications

Colloque

- Thomas Megel, Vincent David, Damien Chabrol, and Christian Fraboul. Dynamic Scheduling of Real-Time Tasks on Multicore Architectures, Colloque du GdR Soc/SiP, 10-12 juin 2009, Orsay.

Conférences

- Thomas Megel, Renaud Sirdey, and Vincent David. Minimizing Task Preemptions and Migrations in Multiprocessor Optimal Real-Time Schedules. In RTSS'10 : Proceedings of the 31st IEEE International Real-Time Systems Symposium, page 37, 2010.
- Thomas Megel, Mathieu Jan, Vincent David, and Christian Fraboul. Task Migration Mechanisms for Hard Real-Time Distributed Systems. In Proceedings WiP RTAS'11 : 17th IEEE Real-Time and Embedded Technology and Applications Symposium, 2011.
- Thomas Megel, Mathieu Jan, Vincent David, and Christian Fraboul. Evaluation of Task Migration Mechanisms for Hard Real-Time Distributed Systems. RTNS'11 : Proceedings of the 19th International Conference on Real-Time and Network Systems, 2011.

École d'été

- Thomas Megel, Mathieu Jan, Vincent David, et Christian Fraboul. Mécanismes de migration pour les systèmes distribués temps-réel stricts. École d'été temps-réel. Brest, 2011.

Chapitre 2

Besoins, état de l'art et problèmes posés

Sommaire

Nous détaillons dans ce chapitre les différents problèmes que posent l'étude, la conception et la validation de systèmes embarqués temps-réel distribués (ou répartis) pour des applications critiques. Nous décrivons les problèmes généraux rencontrés. Pour se faire, nous établissons quelques constats pour comprendre les besoins nécessaires pour ce type de système. Ces besoins conduisent à formuler différents problèmes : niveau temps-réel, il faut pouvoir assurer la ponctualité des traitements et des communications. Le placement des tâches doit être optimisé suivant des critères liés aux ressources, par exemple l'équilibre de la charge. La sûreté de fonctionnement et plus particulièrement la tolérance aux fautes nécessite des mécanismes de détection, d'isolation et de reconfiguration. Nous montrons comment ces problèmes sont résolus par les approches existantes.

Nous nous intéressons ensuite aux objectifs de cette thèse : rendre plus tolérants et plus performants les systèmes distribués temps-réel critiques. L'allocation des tâches est souvent statique pour ce type de systèmes. La flexibilité peut être néanmoins utile, nous le montrons dans le cadre de la tolérance aux fautes. Ceci nous permet d'introduire la problématique de la conception d'un système flexible pour tolérer des défaillances multiples et permanentes. Sur l'aspect des performances, nous nous concentrons sur l'exécution des tâches temps-réel sur une architecture multicœurs. Les politiques optimales existantes permettent théoriquement la pleine utilisation des capacités de calcul, mais plusieurs surcoûts (changements de contexte, complexité) amènent en pratique à des performances plus faibles. Nous nous intéressons à la problématique de l'obtention d'ordonnancements optimaux minimisant ces surcoûts.

2.1 Constat et description des besoins systèmes

Dans un objectif de maîtrise des coûts de production, les industriels concevant les systèmes temps-réel embarqués préfèrent désormais les composants "sur étagère" COTS (*Commercial Off-The-Shelf*) que la fabrication de composants sur mesure (matériel dédié) qui engendrent des coûts bien plus importants. De même, les topologies de communication traditionnellement employées, comme le point à point ou un bus mono-émetteur ne répondent que partiellement au problème du poids et d'encombrement.

L'augmentation du nombre de fonctionnalités à intégrer dans ces systèmes implique une augmentation en besoin de calcul. Au niveau de la capacité de calcul, la puissance des processeurs est liée à leur fréquence d'horloge et l'augmentation de cette fréquence accroît la densité de puissance (W/m^2). La loi de Moore stipule que le nombre de transistors par unité de surface double tous les dix-huit mois. Actuellement, les fréquences ne peuvent pas dépasser quelques GHz, car cette densité de puissance atteint rapidement ses limites pour les architectures embarquées. L'une des solutions utilisées est de concevoir des architectures distribuées : composées de nœuds de calcul disposant d'un processeur, reliés entre eux par un réseau. Un deuxième niveau de parallélisme est désormais possible avec les MPSoC (*Multi Processors System on Chip*), ils permettent à un processeur de disposer de plusieurs cœurs de calcul.

2.1.1 Besoins des traitements en termes de performance

Les performances vont être directement liées à la capacité d'exploitation du parallélisme. Ce parallélisme se retrouve à un niveau local (intra-nœud) et global (inter-nœud) : les architectures distribuées pour les systèmes embarqués répondent donc à l'objectif de capacité de calcul mais encore faut il pouvoir proposer des modèles d'exécution appropriés. La gestion des traitements doit être *multitâches* et rester efficace lorsqu'un nombre important de cœurs de calculs est impliqué, on parle d'extensibilité (*scalability*). D'une manière générale, les performances sont également liées au domaine applicatif, ici nous nous concentrerons sur le *temps-réel* strict : le bon fonctionnement du système nécessite le respect de contraintes temporelles précises.

2.1.2 Besoins en gestion des ressources

Dans un contexte embarqué, l'architecture dispose d'un nombre limité de ressources. Un effort doit être fait pour fournir par exemple une allocation des tâches respectant différents critères. Ils correspondent à des contraintes liées à l'utilisation des ressources du système. La mémoire permet de stocker le système d'exploitation, les contextes des tâches et leur données. Cette mémoire est répartie entre les nœuds et n'excède pas quelques Mo par nœud. La puissance de calcul est lié à la fréquence horloge et au nombre de cœurs de calcul, actuellement ce nombre peut atteindre la dizaine. Les communications se font via des réseaux pouvant atteindre des débits de l'ordre de 100 Mbits/sec. Il faut donc répartir la charge sur les différents nœuds, tout en utilisant efficacement le réseau. Enfin, l'augmentation des ressources passe aussi par une augmentation de la consommation totale d'énergie du système (problématique dans un contexte embarqué).

2.1.3 Besoins en sûreté de fonctionnement

L'augmentation de la puissance de calcul est aussi un moyen d'introduire de nouvelles fonctions dans les systèmes tout en gardant la même maîtrise de fonctionnement qu'auparavant : c'est indispensable pour ce type de système, tels les systèmes *X-by-wire* évoqués précédemment qui viennent remplacer un composant mécanique essentiel à la sécurité. Ces nouveaux systèmes

doivent répondre aussi à des exigences de *sécurité* pour assurer la *cohabitation* de fonctionnalités critiques et non critiques, dont le fonctionnement est optionnel. Nous nous intéresserons uniquement à la prise en charge de tâches critiques. De part leur présence dans des environnements critiques, ces systèmes nécessitent un haut niveau de *disponibilité* (le fait d'assurer la continuité de service) et de *fiabilité* (le fait d'assurer un fonctionnement correct du système, c'est-à-dire fidèle à son fonctionnement prévu).

Il faut pouvoir prévenir des dégradations physiques, tolérer les SoC qui font l'objet de défaillances dès leur fabrication (problème de rendement) ou pendant leur fonctionnement (vieillissement par exemple). L'utilisation de composants matériels sur étagère COTS conduit en plus à des pannes plus fréquentes et de nature plus complexe. De même, la partie logicielle du système (OS, programmes) n'est pas exempte de fautes.

2.1.4 Besoins en reconfiguration

La reconfiguration est la capacité du système à s'adapter et s'applique par exemple à la tolérance aux fautes ou aux changements de modes. Niveau *tolérance aux fautes*, cette capacité est utile pour éviter la production d'un nouveau SoC (affecté dès sa fabrication par des problèmes de fiabilité) ou pour différer un remplacement lors d'une maintenance : il est alors nécessaire d'adapter en conséquence le fonctionnement du système. Au niveau fonctionnel, il faut pourvoir s'adapter à différentes phases de fonctionnement d'un système, ce qu'on appelle les *changements de modes* : par exemple l'initialisation, un fonctionnement en basse consommation. Enfin, il peut être intéressant de gérer l'ajout de fonctionnalités ou la montée en charge du système.

2.2 Problèmes posés par la conception de systèmes embarqués temps-réel répartis

Parmi les problèmes de la conception de systèmes embarqués temps-réel critiques, nous nous intéressons dans cette section à quelques problèmes généraux d'une part, puis nous présentons un état de l'art sur le sujet.

2.2.1 Problèmes fondamentaux

Les problèmes liés aux performances concernent l'efficacité de l'exécution des tâches du système, le respect du temps-réel et la capacité à exploiter le parallélisme local (intra-nœud) et global (inter-nœud). Les systèmes embarqués imposent des contraintes sur les ressources, qui sont en nombre limité et demandent une gestion stricte et déterministe.

Les tâches ont des besoins de calcul et de communication, il sera intéressant d'obtenir un **placement optimisé**. Les ressources sont réparties sur les différents nœud et comme elles sont limitées, il s'agit de résoudre des problèmes d'affectations multi-critères (mémoire, CPU, réseau). Par exemple, on peut considérer que les communications intra-nœud sont moins coûteuses qu'inter-nœud. Comment trouver l'allocation la plus équilibrée en terme de charge moyenne des CPUs minimisant les communications inter-nœud ?

La **ponctualité** est une propriété très importante dans un système temps-réel strict. Elle définit le respect des contraintes temporelles des traitements, notamment la vérification de non-dépassement des échéances. Dans un système critique, un tel dépassement pourrait avoir des

conséquences graves.

Outre le placement, l'ordonnancement devra être **valide** et **ponctuel** i.e. une même tâche ne doit pas s'exécuter simultanément sur plusieurs processeurs en même temps et celle-ci doit respecter ses échéances. La question de la gestion de la priorité des tâches se pose (statique, dynamique) et de la stratégie d'exécution employée (hors-ligne, en-ligne). Le système doit pouvoir exécuter les tâches en parallèle (**multitâches**) au niveau local et global. Par exemple, un problème central consiste à savoir comment assurer la ponctualité des communications locales et des communications distantes (inter-nœud) dépendantes des délais de transmission.

Les systèmes critiques demandent des garanties en terme de sûreté de fonctionnement car des défaillances impactant les tâches peuvent avoir des conséquences au niveau financier ou humain. Une haute fiabilité et disponibilité demande une tolérance à de multiples défaillances qui doit être adaptée au système ciblé.

La **sûreté de fonctionnement** (*dependability*) est l'aptitude d'une entité à satisfaire à une ou plusieurs fonctions requises dans des conditions données [Vil88]. Celle-ci s'évalue en fonction de différents attributs, en voici quelques uns : disponibilité, fiabilité, sécurité, intégrité, survivabilité, sûreté et maintenabilité. Tous ces attributs vont pouvoir s'évaluer en fonction des mécanismes logiciels ou matériels mis en place. Ces mécanismes sont nécessaires pour parvenir à prévenir, tolérer, éliminer ou même prévoir les défaillances. Chaque défaillance peut provenir d'une ou de multiples *fautes* : une défaillance est en fait le résultat d'une *faute* d'origine interne au système ou externe. Cette faute se manifeste comme une *erreur*, un état anormal du système.

Nous avons vu que le placement des tâches sur les unités de calcul doit être optimisé. Avec la nécessité d'apporter un système sûr de fonctionnement, cette allocation doit être fait de telle manière à garantir une utilisation bornée des ressources mémoire, réseau et charge CPU. En effet tout dépassement de ces ressources peut avoir des conséquences importantes. Il s'agit alors d'apporter une **preuve de dimensionnement** suite à l'allocation des tâches sur les unités de calcul, i.e. la preuve qu'en toute circonstance, l'utilisation des ressources restera bornée.

L'efficacité des mécanismes mis en place dépend très fortement de la précision du **modèle de fautes** et de la couverture pratique des mécanismes mis en place. Le modèle de fautes permet de définir des hypothèses sur la nature précise des comportements observables suite aux défaillances correspondant aux fautes possibles subies par le système. L'intérêt d'un modèle précis a été démontré par Latronico [Lat05] sur des mécanismes de synchronisation d'horloges et d'appartenance de groupes (*membership*) : dans la plupart des cas, la probabilité de défaillance est dominé par un type particulier de fautes ; en identifiant cette cause, on peut utiliser un algorithme plus approprié ce qui augmente significativement la fiabilité du système.

Le choix d'un modèle de fautes approprié par rapport aux fautes attendues permet de construire les mécanismes les plus efficaces. Beaucoup de classifications existent, chacune de ces classes demande un traitement spécifique de tolérance aux fautes. Les fautes de conception logicielle ne peuvent être tolérées par simple réplication matérielle, mais demande d'autres stratégies logicielles. Par exemple, la diversification logicielle peut s'effectuer par une méthode de développement de plusieurs versions du code d'un programme par des équipes différentes à partir des mêmes spécifications (i.e. le *N-version programming* [AK84]). A un modèle faute peut être associé des hypothèses de fautes maximales (*Maximum Fault Assumptions*) pour chaque service du système (algorithme de synchronisation, de communication fiable, etc..), ceci permet-

tant d'évaluer les attributs de la sûreté de fonctionnement [Pow95].

Nature des défaillances Dans les systèmes distribués, l'un des plus anciens modèles est celui de Lamport [LSP82], le modèle Byzantin, distinguant entre fautes bénignes (uniformément vues par tous les acteurs du système de manière évidente) et les fautes malicieuses (dont le comportement observable diffère d'un acteur à l'autre). Depuis, d'autres modèles ont vu le jour, notamment les modèles *hybrides* s'affinant de plus en plus avec celui d'Azadmanesh [AK00]. Cette terminologie présente cinq catégories et nous illustrons leur nature sur l'exemple d'un système distribué composé d'un nœud émetteur et de plusieurs récepteurs reliés ensemble par un bus dont les communications sont dirigées par le temps, et l'accès au bus est découpé en plage temporelle : il s'agit d'un bus *time-triggered*. Le classement ci-dessous présente les natures des fautes en fonction de la facilité à les tolérer :

$\mathcal{B}$ fautes bénignes : uniformément détectées par tous les acteurs du système, par exemple l'émission d'un message explicitement erroné (mauvais format de trame) ;

$\mathcal{T}_\mathcal{S}$ fautes par transmission symétrique : par exemple l'émission d'un message non explicitement erroné mais dont le contenu est incorrect ;

$\mathcal{O}_\mathcal{S}$ fautes par omission symétrique : par exemple, le crash silencieux d'un nœud censé émettre sur le slot temporel lui étant réservé ;

$\mathcal{O}_\mathcal{A}$ fautes par omission asymétrique : souvent considérées comme byzantines, elles peuvent néanmoins être tolérées plus facilement comme l'ont montré Serafini et al. [SBS07] avec un consensus hybride à votes majoritaires. Par exemple, les fautes *Slightly Off Specification* sont de ce type [Ade02] : lorsque dans un réseau synchrone un message correct est émis à la limite de sa zone d'émission. Selon la précision des horloges des nœuds certains pourront considérer le message comme correct (car leur horloge est légèrement en retard) alors que d'autres le considéront comme fautif ;

$\mathcal{T}_\mathcal{A}$ fautes par transmission asymétrique (byzantine) : un message erroné est transmis de manière différente aux récepteurs du message. C'est le cas le plus grave, elles peuvent conduire à des émissions incontrôlées et/ou malicieuses. Certains comportements peuvent être identifiés comme le bavardage (*jabbering, babbling idiot*) et dans notre exemple, cela conduit nécessairement à un mode commun de défaillance.

Persistance des défaillances La persistance des défaillances est aussi un critère important. Les défaillances qui persistent au cours du temps sont considérées comme *permanentes* alors que celles dont la défaillance est d'une durée limitée dans le temps sont dites *transitoires*. Si la tolérance n'est pas adaptée, on peut se retrouver rapidement avec un système défaillant.

En résumé, la conception d'un système critique distribué intégrant des composants COTS doit proposer un processus complet, parfois nommé FDIR (*Fault Detection followed by Isolation and Reconfiguration*) :

- la **détection** consiste à identifier un état erroné d'un des composants (logiciels ou matériels) du système. Le système doit pouvoir distinguer entre défaillances de type permanente ou transitoire, de comportement silencieux ou non, et enfin dont la manifestation est de nature symétrique ou asymétrique (selon le modèle approprié) ;
- l'**isolation** permet de localiser par un diagnostic au niveau global et de manière uniforme le composant fautif et de le confiner. Il faut s'assurer que les composants du système (tâches

critiques, OS) puissent continuer leur exécution sans être affectés par un comportement non prévu des autres composants du système. L'isolation doit pouvoir être faite entre composants critiques et non-critiques et si possible entre eux. De par la nature distribuée du système, les décisions devront s'effectuer par *consensus* ;

- la **reconfiguration** est l'application de la stratégie adéquate et plus particulièrement des techniques de recouvrement. Les défaillances impactent une partie du système, des *recouvrements* doivent permettre de les tolérer afin d'assurer la continuité de service (*disponibilité*).

Enfin, les mécanismes nécessaires doivent fournir un comportement garanti conforme aux spécifications du système. Leur exécution doit être *efficace*, réagissant conformément aux spécifications. Car un système où les défaillances ne sont pas détectées ou corrigées à temps peut avoir des conséquences graves. Ces spécifications peuvent être par exemple de garantir une exécution bornée et un comportement déterministe, amenant le système dans un état connu et cohérent. Une exécution bornée permet notamment l'évaluation de la fiabilité et de la disponibilité du système.

2.2.2 État de l'art pour la conception de systèmes temps-réel embarqués

Nous décrivons ci-dessous les solutions existantes au niveau théorique et industriel. Cette section se découpe en deux parties : la première présente les solutions qui répondent aux questions soulevées sur la ponctualité, l'allocation et sur les stratégies d'exécution de ces systèmes et leur flexibilité. La deuxième présente les solutions existantes pour fournir la tolérance aux fautes dans les systèmes temps-réel embarqués.

2.2.2.A. Communications, stratégies d'exécution et flexibilité

Nous allons voir quelles sont les principales approches pour prouver l'ordonnançabilité de tâches communicantes dans un système distribué : elles ne demandent pas toutes l'obtention d'un ordonnancement statique. Nous verrons ensuite quelles sont les stratégies d'exécution des systèmes temps-réel dans l'état de l'art académique et industriel. Enfin, nous décrirons les approches permettant la flexibilité de conception et la flexibilité opérationnelle.

2.2.2.A-1. Communications temps-réel dans un système distribué

Les analyses d'ordonnancement multiprocesseurs classiques ne sont pas directement applicables aux systèmes distribués. Pour garantir des communications temps-réel, outre le placement des tâches que nous détaillerons au début du chapitre 3, il faut vérifier *conjointement* la ponctualité de l'exécution des traitements et des communications de manière globale. Dans cette optique, plusieurs méthodes d'ordonnançabilité conjointes CPU et réseau ont été proposées. On trouve dans l'état de l'art principalement trois approches pour vérifier l'ordonnançabilité : l'approche *statique*, l'approche *holistique* et l'approche *compositionnelle*.

La première méthode construit le plan d'ordonnancement processeur et réseau statiquement. Dans MARS [KDK+89], on sépare les problèmes de placement et d'ordonnancement des tâches, ce qui peut conduire à des solutions sous-optimales. Les deux problèmes ont cependant été traités globalement par Abdelzaher et Shin [AS99], Xu et Parnas [XP93], en introduisant les communications comme des contraintes supplémentaires (relations de précédence) pour placer les tâches sur les processeurs et le plan d'ordonnancement est construit sur chaque processeur avec EDF [LL73]. Un algorithme de *Branch and Bound* est utilisé pour trouver une solution

valide. La complexité de l'approche a conduit à proposer des heuristiques mais l'inconvénient principal est bien sûr la quantité de mémoire nécessaire pour stocker le plan d'ordonnancement de tous les CPUs et du réseau.

La deuxième approche dite holistique [1] [TC94, PGH98] étend les théories classiques prenant en compte un modèle de tâches périodiques appelé *transaction* : il repose sur la modélisation des dépendances de tâches lors de communications distantes (messages) par l'introduction de gigue et d'offset. Cette propriété d'indépendance des tâches facilite l'analyse, le raisonnement peut alors se faire processeur par processeur. La recherche du placement peut se faire par des heuristiques, comme la méthode par recuit simulé de Tindell [TC94]. Même si initialement, les études se sont concentrées sur une politique d'ordonnancement à priorité fixe, rien n'empêche l'analyse sur d'autres politiques, potentiellement plus efficaces. Le principe de l'analyse se base sur le calcul du pire temps de réponse pour chacune des tâches, c'est à dire l'évaluation d'une borne supérieure de la date de fin de traitements en considérant toutes les *interférences* [2] possibles avec les autres tâches du même nœud.

La troisième approche, dite compositionnelle [TCN00] vise à faire l'analyse d'ordonnançabilité de manière modulaire, selon une méthode appelée *real-time calculus*. Celle-ci est basée sur le *network calculus* [Cru91] qui a permis d'établir des bornes supérieures des délais de bout en bout dans les réseaux industriels comme l'AFDX [CSEF06]. L'approche *real-time calculus* permet de coupler les techniques d'ordonnançabilité locales sur des composants via des flux d'évènements définis par des courbes d'arrivée ou de services. Ce modèle s'accommode bien avec des applications temps-réel souples telles que l'encodage/décodage de flux MPEG par exemple.

Ces approches répondent donc à la problématique du respect de la ponctualité sur un système distribué avec des contraintes temps-réel strictes, chacune ayant ses propres caractéristiques et s'adaptant plus ou moins bien en fonction des applications et réseaux embarqués temps-réel considérés.

2.2.2.A-2. Stratégies d'exécution

Nous allons voir maintenant comment les systèmes envisagent l'allocation et l'ordonnancement des tâches critiques dans un contexte temps-réel embarqué.

Un ordonnancement est dit **en-ligne** lorsque les décisions d'ordonnancement se font lors de l'exécution des tâches et **hors-ligne** lorsque les décisions sont prises lors de la compilation par exemple. Un ordonnancement hors-ligne conduit à une exécution statique, tandis qu'un ordonnancement en-ligne conduit à une exécution dynamique.

D'une manière générale, une exécution statique est souvent employée pour l'exécution des tâches temps-réel *critiques*, car la vérification de leur ponctualité est facile. Le projet MARS par exemple se base sur une analyse statique pour démontrer que les contraintes sont respectées. Un ordonnancement hors-ligne est alors produit et écrit dans des tables lues lors de l'exécution. Une autre approche, Spring [SR89], permet l'exécution de tâches critiques, également exécutées de manière statique, avec d'autres tâches gérées dynamiquement. Ces dernières pouvant être migrées d'un nœud à l'autre. La norme Autosar pour la conception automobile se base également

1. Du grec holos, "le tout", la totalité
2. les interférences sont définies dans un contexte à priorité, comme l'ensemble des tâches de priorité supérieure active au moment de l'exécution de la tâches concernée

sur la description de tables d'ordonnancement (*schedule tables*) pour décrire des comportements statiques et cycliques des tâches [eHmDFT07].

Néanmoins, il est possible de considérer une exécution dynamique en garantissant hors-ligne les propriétés d'exécution, c'est l'approche exploitée dans OASIS [AD98] qui propose une méthodologie de développement et une chaîne d'outils originale pour la conception et la réalisation de systèmes temps-réel strict.

Comme noté par Stankovic et al. [SR04], les systèmes temps-réel sont souvent conçus autour d'un ordonnancement particulier. La plupart des RTOS commerciaux embarqués se basent sur l'approche à priorité fixe et par conséquent sur les techniques d'ordonnançabilité correspondantes. C'est le cas d'OSEK, un standard pour la conception de systèmes d'exploitation temps-réel pour l'automobile ou du profil Ravenscar conçu pour le développement d'applications temps-réel déterministes et "efficaces" (le profil apporte des restrictions définissant un sous-ensemble des mécanismes utilisés dans le langage ADA). Même si ce choix peut paraître surprenant par rapport aux performances globales (voir chapitre 5), l'une des raisons de ce choix est la facilité d'implémentation pour les noyaux ne disposant pas de support pour les contraintes temps-réel.

Dans une grande majorité des cas, l'allocation des tâches, de leur réplique et leur ordonnancement est souvent statique sur des systèmes embarqués critiques. Il faut donc choisir l'allocation la plus adaptée au besoin du système. Comme évoqué précédemment, il s'agit de problèmes d'affectations multi-critères. Un compromis doit être effectué entre les allocations possibles suivant différentes métriques d'évaluation.

2.2.2.A-3. Flexibilité de l'allocation dans les systèmes embarqués temps-réel

Différents frameworks ont été proposés pour concevoir et assigner les tâches dans les systèmes embarqués fiables [IPEP05, JKH05, LKY+00, SJH+10]. Ces méthodes proposent différentes heuristiques et différentes manières d'évaluer les placements de tâches obtenus suivant des paramètres multicritères : ceci permet une grande flexibilité dans le choix des allocations. Un compromis peut être fait pour trouver une solution, on parle de **flexibilité de conception**. Par exemple, on cherche une solution Pareto-optimale. Une solution est **Pareto-optimale** lorsqu'il n'existe pas d'autres solutions faisables telles que la diminution d'un objectif n'ait pour conséquence l'augmentation d'au moins un autre objectif[3].

Comme énoncé précédemment, le concepteur obtient en général un placement statique *unique* sur l'architecture matérielle. Ces approches répondent donc au besoin concernant l'allocation suivant plusieurs critères comme l'équilibre de la charge, la minimisation des communications distantes. Nous allons voir maintenant les approches pour la *flexibilité opérationnelle* [AFAL07] afin de voir quels avantages elles apportent.

La **flexibilité opérationnelle** se traduit par la capacité à s'adapter aux changements en-ligne : il peut s'agir d'une politique d'allocation dynamique, de la gestion de changements de modes ou pour la tolérance aux fautes.

Cette flexibilité est souvent réservée aux domaines *best-effort*, effectuée dynamiquement. L'approche de Ramamritham et Stankovic [RS84] dispose d'une politique de décisions pour l'ac-

3. Plus formellement, cela correspond à la recherche des éléments minimaux des ensembles partiellement ordonnés (poset) des solutions.

ceptation et l'allocation des tâches non-critiques : celles-ci sont réparties et migrées en fonction des ressources disponibles sur les différents nœuds du système.

Les changements de modes correspondent à des phases de fonctionnement du système. Cela se traduit au niveau logiciel par un changement de tout ou partie de l'ensemble des tâches en cours d'exécution. Ce changement se traduit soit par des temps d'exécution différents des tâches, soit par un remplacement par de nouvelles tâches avec des caractéristiques temporelles différentes (échéances, loi d'arrivée, etc.). Des protocoles de changement de modes temps-réel existent et permettent de vérifier l'ordonnançabilité en monoprocesseur. Real et al. [RC04] propose un *survey* des techniques monoprocesseur à priorité fixe. Elles sont classées suivant différents critères : le respect des échéances des tâches de l'ancien mode, la gestion de tâches indépendantes des modes, la promptitude du changement de mode (son délai) et la consistance i.e. le respect de l'utilisation des ressources partagées. Récemment, Borde a développé dans sa thèse [Bor09] une méthodologie de conception pour des systèmes embarqués temps-réel répartis monoprocesseur s'adaptant à différents modes à l'aide des techniques existantes. Enfin, Nelis et al. [NGA09, NAG09]) ont proposé des extensions pour multiprocesseurs.

Il existe peu de travaux sur l'allocation en-ligne pour la fiabilité et l'équilibre de la charge suite aux défaillances, nous les verrons au chapitre 3. Aucune de ces approches ne s'est intéressée à l'utilisation de répliques logicielles pour la tolérance aux fautes. Nous présentons cette problématique à la section suivante et la détaillerons au chapitre 3.

Nous avons vu la manière dont est gérée l'exécution sur les architectures embarqués temps-réel orienté sûreté et de quelle flexibilité elles disposent. Il reste à voir quelles approches de réplication logicielle sont envisageables, et comment ces méthodes permettent de garantir la ponctualité des traitements en présence de défaillances par la génération d'un ordonnancement approprié et comment il est possible de s'assurer de la cohérence de leur exécution.

Comme indiqué initialement dans les besoins, un système critique ne dépend pas uniquement de ses performances mais également la confiance que l'on peut lui donner. La *sûreté de fonctionnement* permet d'évaluer cette confiance et nous allons donc présenter les mécanismes permettant de répondre à cette problématique.

2.2.2.B. Tolérance aux fautes

Nous nous concentrons principalement sur les approches liées à la tolérance aux fautes dans un contexte embarqué, limité en ressources. Néanmoins, on ne peut parler de tolérance aux fautes sans évoquer la sûreté de fonctionnement qui l'englobe et comment elle permet de répondre aux problèmes posés précédemment. Nous essaierons dans les sections suivantes de présenter ces mécanismes dans les projets académiques ou industriels en précisant, lorsque cela est possible, la nature des fautes suivant la terminologie d'Azadmanesh et al. [AK00] présentée à la section précédente p. 17.

2.2.2.B-1. Les mécanismes de détection et de confinement

Ces deux mécanismes doivent être reliés à un *consensus* dès lors que l'on considère une architecture distribuée à composants COTS présentants des fautes de natures asymétriques. Le consensus est un mécanisme permettant de s'accorder sur une valeur par exemple. Au niveau système, il faut s'accorder sur la *vue du système*, c'est à dire sur le comportement normal ou

défaillant de chacun des composants (matériels ou logiciels) du système. Pour le niveau système [4] nous parlerons de *diagnostic distribué*. Les meilleurs algorithmes de consensus décident en temps polynomial de manière déterministe [GM98], c'est à dire en obtenant un résultat au bout d'un temps borné (dépendant du nombre de défaillances byzantines simultanées) : c'est un prérequis pour des systèmes critiques où les mécanismes de tolérance doivent pouvoir être réactifs. Des diagnostics distribués rapides existent désormais pour des modèles de fautes plus précis englobant des comportements asymétriques ($\mathcal{O}_{\mathcal{A}}$) [SBS07]. Enfin, on définit l'*unité de confinement* c'est à dire la plus petite partie que l'on peut isoler lors d'une défaillance sans implication sur le reste du système, qu'elle soit logicielle ou matérielle. Voici quelques projets et réseaux proposant ce type de mécanismes.

Projets de recherche Les systèmes fiables temps-réel académiques ont fait l'objet de nombreux projets, parmi ceux-là : dans l'ordre chronologique DELTA-4 (Powell et al., 88), MARS (Kopetz et al., 89), GUARDS (Powell et al., 99) et DEAR-COTS (Verissimo et al., 2000-04). Toutes ces architectures ont tenté de définir des modèles de conception d'applications temps-réel fiables, fournissant des mécanismes de tolérance aux fautes.

DELTA-4 [PBS+88] est projet issu du programme européen ESPRIT. Il propose une architecture où les nœuds sont distribués par un réseau fibre (FDDI) dont le fonctionnement est proche de l'anneau à jeton. Ces nœuds sont divisés en deux entités : l'hôte (composant COTS) supposé défaillir de manière arbitraire ($\mathcal{T}_{\mathcal{S}}$) mais relié à un contrôleur réseau (NAC) qui lui est spécifique et supposé à défaillance silencieuse (*fail-silent* $\mathcal{O}_{\mathcal{S}}$) par auto-surveillance matérielle (*self-checking*).

MARS [KDK+89] (MAintainable Real-time System) est un projet développé à l'université technologique de Vienne qui propose des calculateurs trois fois redondés, connectés à deux bus. Chaque calculateur dispose de deux processeurs l'un pour la partie applicative, l'autre pour les communications. L'hypothèse sur le modèle de fautes est l'arrêt des calculateurs de manière silencieuse (modèle de faute par omission symétrique $\mathcal{O}_{\mathcal{S}}$). L'unité de confinement appelé FTU (Fault Tolerant Unit) est à la granularité de groupes de trois calculateurs (redondance triple TMR). Les deux bus ne transmettent que des défaillances par omission symétrique ($\mathcal{O}_{\mathcal{S}}$), donc à priori aucun consensus n'est nécessaire pour déterminer quel message issu d'une FTU doit être utilisé. TTA [KG94] (Time-Triggered Architecture) est une architecture issue du projet MARS, qui comme son nom l'indique se base sur des réseaux dirigés par le temps. Autre différence, la présence des "pare-feux", composants dédiés sur chaque nœud pour garantir le comportement silencieux suite à défaillance.

Certaines approches ont privilégié des réseaux dédiés pour apporter la détection et le confinement des défaillances. Dans tous les cas, une architecture parallèle de nœuds reliés par un réseau permet déjà une isolation en cas de dégradation physique.

GUARDS [PABD+99] (A Generic Upgradable Architecture for Real-Time Dependable Systems) est un projet du programme européen ESPRIT. Les hypothèses sur le modèle de fautes sont moins restrictives et adaptables. L'architecture est structurée en canaux parallèles composés de deux à quatre calculateurs, voir schéma Fig. 2.1. Ces calculateurs sont reliés entre eux par un composant dédié inter-canal implantant au niveau matériel un mécanisme de consensus pouvant s'adapter suivant la redondance, au silence sur défaillance ($\mathcal{O}_{\mathcal{S}}$) avec deux bus ou à l'apparition

4. les protocoles pour la synchronisation d'horloges sont également basés sur le principe du consensus, même si les valeurs obtenues sont des valeurs approchées

de fautes byzantines ($\mathcal{T}_\mathcal{A}$) avec trois bus.

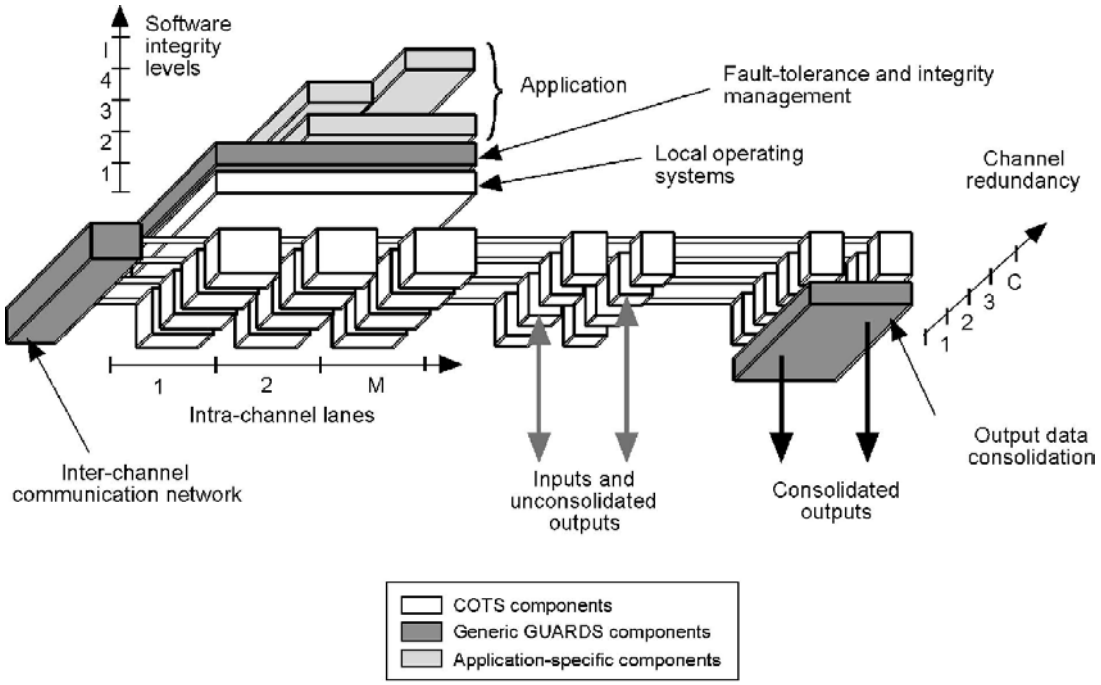


FIGURE 2.1 – Description générale de l'architecture matérielle et logicielle GUARDS, schéma issu de [PABD+99].

Dans l'aérospatiale (NASA), le projet SPIDER [MMT02] (Scalable Processor-Independent Design for Electromagnetic Resilience) exploite une architecture sur mesure de type point à point entre des unités de gestion de la redondance (RMU) et des interfaces de contrôle nommées BIU (Bus Interface Unit) elles-mêmes en lien direct avec les unités de calcul (PE). Cette topologie particulière, voir Fig. 2.2 (appelée ROBUS Reliable Optical Bus) permet une tolérance à des fautes de nature très différente (symétriques $\mathcal{O}_\mathcal{S}, \mathcal{T}_\mathcal{S}$ et asymétrique $\mathcal{T}_\mathcal{A}$) via l'utilisation d'un protocole de consensus unifié pour plusieurs mécanismes : la synchronisation des horloges, le diagnostic distribué et les communications fiables.

Plus récemment BRAIN [PH07] (Braided Ring Availability Integrity Network) propose une architecture matérielle sous forme d'un anneau tressé à base de composants Ethernet apportant de bons résultats en terme de fiabilité avec des capacités d'auto-surveillance (*self-checking*) matérielle et de confinement très fine par l'utilisation du protocole réseau TDMA (dirigé par le temps). Le modèle de fautes hybride envisagé permet de tolérer en autres la présence d'une défaillance arbitraire ($\mathcal{T}_\mathcal{S}$).

Réseaux Industriels Parmi les réseaux fournissant des communications temps-réel critiques dans un contexte embarqué, rappelons SAFEBus pour Boeing 777, l'architecture TTA (Time-triggered Architecture) pour le système de communications de l'A380 Dreamliner commercialisé par TTTech [TTT] et l'AFDX (Avionics Full-Duplex Switched Ethernet) pour Airbus A380.
SAFEbus [ARI93] est une implémentation par Honeywell du standard ARINC 659. La topologie est un bus redondé et dont chacun des nœuds dispose de deux contrôleurs réseau s'auto-

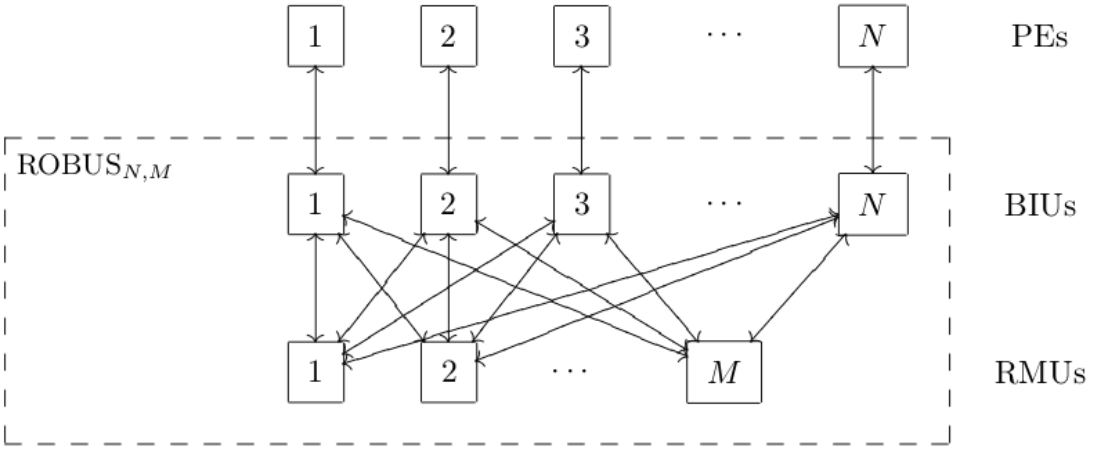


FIGURE 2.2 – L'architecture de communication ROBUS du système SPIDER, schéma issu de [MMT02].

surveillant. Les données étant transférées de manière prédéterminée, une détection et isolation très rapide est possible. Cette détection bit à bit limite néanmoins son débit, qui atteint 60 Mbits/sec maximum. Cette architecture permet une tolérance à une défaillance unique de toute nature. SAFEbus et TTA permettent la détection et l'isolation de communications non désirées comme le *jabbering* ou le *Babbling Idiot*.

Si TTA et SAFEbus sont dits déterministes permettant une détection très fine, l'AFDX [ARI02] filtre les communications uniquement par rapport à la charge réseau utilisée. Niveau isolation, les architectures TTA et SAFEBus utilisent des composants dédiés pour confiner les calculateurs défectueux, alors que l'AFDX utilise des commutateurs qui sont configurés pour couper toute charge réseau supplémentaire. Un comparatif sur les différentes caractéristiques de certains réseaux sont disponibles dans le rapport technique de Rushby [Rus01].

Même si nous nous concentrerons sur les défaillances matérielles, la partie logicielle n'est pas exemple de fautes que l'on doit d'une manière similaire détecter et confiner. Au niveau logiciel, l'isolation du comportement des tâches est reliée à la sécurité. Pour plus de détails, le lecteur peut consulter certains principes de conception pour des systèmes sécurisés de Saltzer et Schroder [SS75]. De même le contrôle de l'accès aux ressources peut se faire statiquement (l'approche MILS [AfHOT06] permet l'écriture de systèmes fondés sur une approche distribuée de la sécurité) ou dynamiquement (par *capacité* [Den76]).

Globalement dans ces différents projets, l'unité de confinement considéré, i.e. la plus petite partie que l'on peut isoler lors d'une défaillance, est à la granularité du calculateur (ou du nœud), excepté pour DEAR-COTS qui propose une gestion plus fine au niveau de la tâche. Lorsque la détection n'est pas faite au niveau matériel ou que l'on veut utiliser des composants COTS, la réplication logicielle peut jouer ce rôle, avec l'utilisation de répliques actives. Nous allons donc détailler les stratégies de réplication puis la manière de garantir leur cohérence.

2.2.2.B-2. Les mécanismes de réplications

Les mécanismes de reconfiguration sont principalement des mécanismes de recouvrement

suite aux défaillances. L'utilisation de matériels dédiés pour la détection, la redondance a été couramment employée avec les systèmes embarqués critiques. Avec l'utilisation de composants COTS moins coûteux, la puissance de calcul est en nette augmentation, permettant la mise en place de *redondance logicielle*.

Les stratégies les plus connues sont [PABD+99] : réplication active, semi-active ou passive. La réplication active permet le masquage des défaillances les plus graves si les répliques émettant un message sont soumises à un vote par exemple. Le recouvrement est donc immédiat. C'est stratégie est aussi celle qui demande le plus de ressources. La réplication passive trouve son intérêt dans son utilisation faible des ressources. Les copies sont synchronisées à la tâche active par l'envoi périodique de données. Enfin, la réplication semi-active est une combinaison des deux premières, permettant à une réplique de suivre l'exécution de la tâche active de manière silencieuse. Les avantages et inconvénients peuvent s'évaluer selon différents critères : la nature des fautes détectées, tolérées, le surcoût d'exécution qu'implique chacune des techniques, leur temps de reprise, etc. Pour plus de détails, on peut se référer à l'état de l'art effectué à ce sujet par Kalla dans sa thèse [KAL04].

Même si nous ne nous intéresserons qu'à la tolérance aux défaillances permanentes, il est tout de même intéressant de citer les travaux concernant les défaillances transitoires à la vue des causes possibles de perturbations dépendantes de l'environnement (effets électromagnétiques, effets ionisants dans l'avionique SEU [DM03]). Beaucoup d'études ont été réalisées sur ce sujet, nous présentons brièvement quelques travaux afin d'illustrer les différentes approches dans la section suivante.

La réplication pour fautes transitoires
L'apparition de défaillances transitoires peut être tolérée en prévoyant au niveau de l'ordonnancement les ressources nécessaires pour réexécuter (ou retransmettre) les tâches (ou messages [TBEP10]) impactées à un point dans le temps précédent l'apparition de l'erreur : soit l'apparition suit une distribution probabiliste (Poisson ou Weibull [CMS82]), soit un temps minimal entre deux apparitions fixé à priori. Deux techniques sont majoritairement étudiées :

- par blocs de recouvrement (*recovery blocks*), c'est à dire des traitements redondants d'une même tâche (avec potentiellement des temps d'exécution différents) [TS94, Ayd07] ;
- par points de reprise (*checkpoint*), c'est à dire en prévoyant des points de sauvegarde uniformément ou non réparti au long de l'exécution des tâches.

Même si une majorité des études portent sur la priorité fixe [PBD01, TS94], certaines études ont également été faites pour la priorité dynamique de tâches avec blocs de recouvrement [Ayd07]. Notons également la thèse de Kalla [KAL04], proposant une méthodologie, proche de la méthodologie AAA (Syndex [GLS99]), pour le placement et l'ordonnancement de tâches dépendantes sur un réseau non complètement connecté, tolérant aux fautes transitoires via des répliques passives ou hybrides. Cette approche permet d'envisager n'importe quelle combinaison de fautes transitoires de processeurs ou des liens de communications. Elle consiste en deux phases : une première étape s'appuie sur un formalisme de graphe pour extraire les spécifications de redondance et des contraintes d'exclusion. Ensuite, une heuristique est proposée pour allouer spatialement et temporellement les composants logiciels sur l'architecture cible.

Nous allons maintenant nous intéresser plus particulièrement aux méthodes de tolérance aux fautes permanentes. Celles-ci impactent définitivement une partie des ressources, entraînant la perte des tâches ou répliques allouées sur la ressource défaillante. Plus précisément nous

présentons les travaux répondant à cette problématique assurant l'allocation spatiale pour la redondance logicielle tolérant aux défaillances permanentes.

La réplication pour fautes permanentes
La réplication spatiale est souvent considéré sur un axe commun de placement et d'ordonnancement. Oh et Son [OS97] ont montré que le problème était NP-complet et ont proposé une heuristique en supposant l'arrêt des processeurs de manière silencieuse basée sur la redondance passive tolérant une défaillance. Le principe est de répliquer deux fois chaque composant logiciel sur des processeurs distincts. Seule la tâche s'exécute et si celle-ci défaille, l'une des répliques prend le relai : elle réexécute le traitement manquant lors de la défaillance de la tâche et poursuit l'exécution en tant que tâche active. Sur le même principe, Qin et al. [QHJ+00] élargissent l'approche avec des composants logiciels avec dépendances de données.

Krishna et Shin [KS86] ont montré qu'à partir d'un placement donné, on pouvait obtenir un ordonnancement redondant. Leur idée est de disposer d'ordonnancement de reprise en cas de défaillance (répliques passives). Ces ordonnancements de reprise sont générés par programmation dynamique en supposant l'existence d'un ordonnancement optimal en présence des répliques passives.

Avec des répliques actives, Dima et al. [DGLS01] présentent un placement et l'ordonnancement correspondant avec une réplication active tolérant les fautes des processeurs et des liens de communications. Ils proposent une heuristique pour le placement et l'ordonnancement sur une architecture matérielle, puis sur les architectures issues de chaque configuration d'une défaillance. Enfin, un placement et ordonnancement global est produit par la combinaison de tous ceux déjà produits.

Hou et Shin [HS94] ont montré comment maximiser la fiabilité d'une exécution (en terme de probabilité de terminer les calculs avant échéance) pour l'allocation de répliques actives et de tâches périodiques avec précédences. Les tâches sont découpées en module. Les modules critiques pour le respect des échéances sont sélectionnés en utilisant une analyse de chemin critique. Ceux-ci sont répliqués et l'ensemble des modules et de leur réplique est alloué et ordonnancé statiquement par un algorithme de *Branch and Bound*.

En multiprocesseurs, Manimaran et al. [MSM98] propose un ordonnancement tolérant à des défaillances multiples de processeurs par l'utilisation de copies de sauvegarde qui permettent un recouvrement arrière. Ils calculent des intervalles de temps disjoints afin de rejouer l'exécution d'une tâche lors d'une défaillance qu'ils intègrent pour tester la faisabilité. La nature dynamique de leur algorithme permet d'envisager un compromis entre performance i.e. le ratio de garantie de ponctualité et le niveau de tolérance.

L'obtention d'une allocation et de son ordonnancement dépendent souvent de la stratégie de réplication mis en place. La séparation entre le mécanisme de réplication des tâches et son ordonnancement a fait l'objet de peu de travaux, citons tout de même la thèse de Mossé [Mos93] pour la réplication active et la thèse de Chevochot [Che99] pour les trois principales stratégies. Le principal intérêt de ces approches est de décrire précisément les contraintes temporelles des traitements et communications propre à chaque stratégie de réplication, ainsi l'ordonnancement peut s'effectuer indépendamment de la stratégie de réplication envisagée.

Nous avons vu quels mécanismes de réplications logicielles pouvaient jouer le même rôle que

des composants matériels dédiés pour la tolérance aux fautes. Il reste à s'assurer de la cohérence des données entre répliques. Ce problème a été résolu de plusieurs façons différentes.

Le problème de la cohérence des répliques Poledna [Pol94] a identifié un problème pour ces mécanismes qui dépendent étroitement du temps. Le "déterminisme" de l'exécution des répliques n'est possible que si la cohérence des données d'entrées des répliques est assurée afin que celles-ci produisent, en l'absence de faute, les même résultats (si le nombre de répliques est supérieur à l'unité). Si ce déterminisme peut être atteint en empêchant l'exécution multitâches comme fait dans MARS ou DELTA-4, des approches moins contraignantes basées sur le marquage temporel des messages [PBWB00] permettent d'obtenir le même résultat. Une date de validité est associée au message, précisant quand le message est valide. Le récepteur d'un message ne pourra lire qu'un message valide arrivé avant sa date de début d'exécution. Cette approche a été mise en place de manière explicite dans GUARDS et de manière transparente dans DEAR-COTS. Bien sûr, cette méthode n'est possible que si une cohérence temporelle est globalement mise en place.

2.3 Problèmes abordés

Nous allons maintenant nous attacher plus particulièrement aux problèmes que nous abordons dans cette thèse. Nous évoquons d'abord la flexibilité des systèmes distribués temps-réel, en quoi celle-ci est importante et peut s'avérer utile pour la tolérance aux fautes. Ensuite nous verrons des aspects liés aux performances, la manière dont les tâches sont ordonnancées au niveau local d'un nœud dans le cadre multiprocesseurs à mémoire partagée.

2.3.1 Flexibilité pour la reconfiguration du système suite aux défaillances

Nous avons vu dans l'état de l'art général à la section précédente que le problème de l'allocation des tâches pour un système embarqué temps-réel critique est résolu en proposant une unique allocation. Comme le font remarquer Almeida et al. [AFAL07], les approches de conception existantes apportent de la flexibilité dans le choix de l'allocation des tâches au moment de la conception du système, donnant plus de liberté : on l'appelle la *flexibilité de conception*. Elles permettent de trouver un bon compromis en évaluant différentes métriques liées à la sûreté de fonctionnement, les performances, l'efficacité.

Cependant, il y a plusieurs avantages à envisager un placement plus flexible, pas forcément dynamique, mais s'adaptant en-ligne aux changements d'état du système. La *flexibilité opérationnelle* définit la capacité du système à s'adapter en-ligne aux changements. Comme indiqué dans l'état de l'art de la section précédente, elle permet d'envisager plusieurs bénéfices en termes d'économie d'énergie, pour des changements de modes ou tout simplement pour la qualité de service (QoS). Nous allons voir en quoi cette flexibilité en-ligne peut être utile à l'allocation des tâches pour la gestion de la tolérance aux fautes lorsque le système fait appel à des répliques logicielles.

Les changements du système peuvent être liés aux défaillances entraînant la perte d'une partie des ressources du système : ces évènements sont imprévisibles mais pas imprévus. Comme les approches classiques de conception n'envisagent qu'une seule architecture matérielle (un ensemble statique de ressources), on se prive d'optimiser l'allocation dans le cadre d'un sous-ensemble du système. Par exemple, la défaillance permanente d'un nœud implique que l'architecture matérielle ne peut désormais fonctionner que sur les nœuds valides restants, c'est à dire

avec des ressources CPU, mémoire inférieures. Il serait intéressant d'optimiser l'allocation pour le système suite à défaillances.

Shelton et al. [She03b] ont montré l'intérêt de cette flexibilité pour améliorer la survivabilité des systèmes, i.e. que la dégradation suite aux défaillances peut être minimisé. Nous l'illustrons sur un exemple simple monocritère ci-dessous. L'**utilité système** est une métrique définie par Shelton [She03a] pour évaluer le niveau de service d'un système soumis à des défaillances. L'utilité d'une tâche dénote l'importance de sa fonction dans le système.

Considérons maintenant l'exemple de l'allocation du jeu de tâches critiques temps-réel suivant, issu de l'exemple d'Emberson et al. [EB08], présenté dans la table 2.1.

Tâche	WCET	Période	Répliques	Utilité
A	7	10	2	0.5
B	7	10	2	0.3
C	2	10	2	0.15
D	2	10	2	0.05

TABLE 2.1 – Ensemble de tâches périodiques avec utilité système.

Le concepteur doit trouver l'allocation plaçant les tâches maximisant l'utilité du système en fonction du nombre de défaillances. Chaque tâche peut disposer d'un maximum de 2 répliques actives consommant les mêmes ressources. L'architecture matérielle est constituée de 4 nœuds et peut subir de 1 à 3 défaillances de nœuds.

Suivant l'allocation initiale, on peut tracer le profil de l'utilité correspondant à chaque allocation statique initiale considérée. La Fig. 2.3 suivante est issue de l'exemple d'Emberson et al. [EB08], elle représente la dégradation de l'utilité pour deux allocations initiales possibles de l'architecture 4 nœuds dans le pire des cas de défaillances.

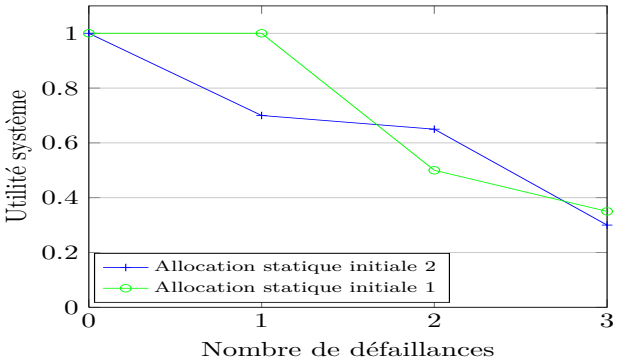


FIGURE 2.3 – Dégradation du système : comparaison de l'utilité système fonction du nombre de fautes et de l'allocation initiale.

Nous pouvons remarquer que selon le nombre de défaillances envisagées un profil est meilleur que l'autre. Comme le remarque Emberson et al., le concepteur doit faire un compromis, qui est réalisable actuellement par l'évaluation d'autres critères, voir Pareto-optimal dans la section 2.2.2 page 20. Cependant, si le système était en mesure de réallouer les tâches et répliques dynamiquement, de manière sûre et bornée à chaque défaillance, il pourrait combiner les possibilités d'allocation pour obtenir à chaque nouveau sous-ensemble du système, la meilleure

allocation possible pour le ou les critères évalués, ici montré sur le seul critère d'utilité système.

Nous proposons donc de mettre en place une approche hors-ligne permettant cette flexibilité en-ligne de manière efficace et sûre pour l'allocation des tâches et répliques s'exécutant sur un système subissant des défaillances multiples. Nous nous intéresserons au niveau global de l'architecture à la manière dont la reconfiguration peut être effectuée en gardant un comportement déterministe et efficace dans le chapitre 3. Nous répondons à différentes questions.

- Un premier problème est celui de modéliser le problème de l'allocation de tâches et de leurs répliques dans le cadre de la tolérance aux fautes multiples et permanentes. Nous modélisons les spécifications des allocations à l'aide d'un langage de programmation linéaire.
- Quelles sont les méthodes efficaces pour obtenir une allocation optimisée respectant les contraintes de ressources ? Nous verrons que la formulation linéaire conduit souvent à la résolution de problèmes NP-difficiles et nous évoquons les alternatives possibles.
- Il est intéressant d'obtenir une allocation optimisée, mais quels critères retenir dans le cadre d'un système subissant des défaillances multiples et permanentes ?
- Lors de la détection d'une défaillance, quelles sont alors les actions à effectuer ? Nous verrons que notre objectif sera d'atteindre une allocation de référence, optimisée pour l'ensemble restant des ressources sur le système.
- Comment s'assurer que tous les scénarios possibles de défaillances sont bien pris en compte ? Nous ferons l'hypothèse d'une architecture symétrique et que nous pouvons alors traiter le problème par niveau de défaillances.
- En optant pour l'exécution de répliques logicielles, comment s'assurer du respect du dimensionnement lors du changement d'activité d'une réplique lors de la perte de la tâche active ? Le changement d'activité correspond au passage de réplique passive au statut de tâche active.

Nous faisons l'hypothèse que la mémoire est une ressource limitée dans les systèmes considérés et qu'il n'est pas possible d'embarquer la totalité de l'espace mémoire des tâches sur chaque nœud. La reconfiguration peut alors demander la réallocation des tâches et répliques, c'est-à-dire de migrer l'espace d'adressage de tâches ou de répliques d'un nœud à l'autre pour atteindre une allocation de référence. Des techniques de migrations appropriées sont alors nécessaires et doivent s'adapter à notre contexte temps-réel strict : un grand nombre de mécanismes de migration de tâches existent pour les systèmes asynchrones mais aucun lorsque le système est soumis à des contraintes temps-réel strictes.

Nous proposons donc de mettre en place dans le chapitre 4 des techniques de migrations adaptées à un contexte temps-réel strict qui permettent notamment des délais bornés et qui ne présentent pas de temps de gel, c'est à dire n'impliquant pas de pause dans l'exécution de la tâche migrante. Autoriser un temps de gel conduirait à un système non prédictible.

- Comment assurer des migrations en temps borné et est-il possible d'éviter le temps de gel ? Nous montrons que cela est possible grâce à la description précise du comportement temporel des tâches et nous proposons différentes techniques évitant le temps de gel.
- De même, il est indispensable de vérifier la faisabilité des migrations : la migration était-elle faisable pendant le temps imparti étant donné les caractéristiques de la tâche migrante et de celles du réseau ? A quoi correspond le temps imparti ?

- Une fois la faisabilité vérifiée, ceci ne permet pas de s'assurer de l'ordonnançabilité globale des tâches du système. En effet, la migration a un coût en terme de ressources et vient s'ajouter à l'exécution des autres tâches du système. Comment vérifier alors l'ordonnançabilité globale du système ?
- Enfin, quel est le surcoût induit par ses migrations sur l'exécution du système ? Comment le mesurer ? Nous répondrons à ces questions et proposons des évaluations par simulation, fonction de plusieurs paramètres comme la taille mémoire de la tâche migrante ou le nombre de migrations.

2.3.2 Performance dans les multicœurs

Nous présentons maintenant des aspects relatifs aux performances dans les architectures multicœurs toujours dans les systèmes embarqués temps-réel stricts.

Pour l'exécution temps-réel sur ces architectures, on doit définir dans le modèle d'exécution quelle stratégie d'ordonnancement est utilisée. L'ordonnancement *global* permet de gérer une liste unique des tâches du système et de prendre les décisions d'ordonnancement de manière centralisé pour l'ensemble des processeurs. A contrario, on parle d'ordonnancement *local par partitionnement* quand chaque ordonnanceur se sert d'une liste locale, chacune d'elles étant indépendante des autres (pas de migration possibles entre tâches de listes différentes). Dans un contexte multitâches, il est possible d'envisager des ordonnanceurs préemptifs, permettant des préemptions i.e. des tâches interrompues puis reprises plus tard, et autorisant ou non des migrations de tâches i.e. des tâches dont l'exécution peut être transférée sur une autre unité de calcul.

Classe d'ordonnanceurs Carpenter et al. [CFH+04] ont permis de classer les différentes politiques d'ordonnancement. Une tâche délivre un nombre infini de travaux, nous considérons qu'un travail est une *instance* d'une tâche. Soit un couple (x, y) / $x, y \in \{1, 2, 3\}$, x définit la **priorité** des tâches ou de leurs travaux :

1. statique (fixe au niveau de la tâche), ex : Rate Monotonic-like (RM [LL73]) ;
2. dynamique mais fixe pour chaque travail, ex : Earliest Deadline First-like (EDF [LL73]) ;
3. complètement dynamique (dynamique au niveau d'un travail), ex : Least Laxity First-like (LLF [DM89]).

et y définit la **capacité de migration** des tâches :

1. partitionné (aucune migration autorisée) ;
2. migration partielle (chaque travail doit s'exécuter entièrement avant de pouvoir migrer) ;
3. migration totale (sans restriction).

Cette classification n'est pas exhaustive car on rencontre désormais des ordonnanceurs *semi-partitionnés* i.e. qui autorisent les migrations uniquement sur des groupes de processeurs (e.g. [KY09]). Néanmoins elle a permis d'identifier selon la classe, le taux d'utilisation U maximal assurant l'ordonnançabilité, se rapporter à [CFH+04] pour plus de détails. Nous pouvons voir sur la Fig. 2.4 un ensemble de 3 tâches périodiques synchrones identiques telles que leur taux d'utilisation nécessite deux processeurs ($U = 2$). Cet ensemble est ordonnançable uniquement sur deux processeurs avec un ordonnanceur de classe (3,3) par exemple en préemptant la première tâche,

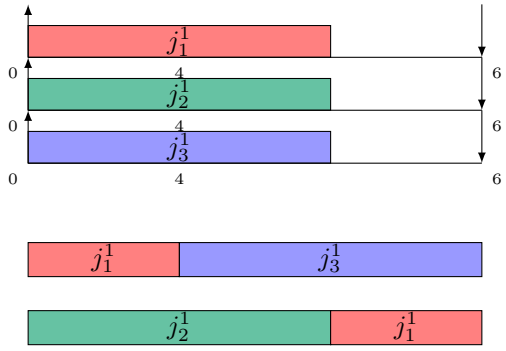


FIGURE 2.4 – Trois tâches périodiques T_1, T_2 et T_3 identiques : leur temps d'exécution est de 4 unités de temps et leur échéance à 6 unités de temps. L'exécution est sur 2 processeurs identiques.

comme montré sur cette figure. *Global-EDF* qui est de classe (2,3) échouerait à ordonnancer cet ensemble de tâches.
Notons que les ordonnancements à priorité fixe donnent des garanties de faisabilité assez faibles en multiprocesseur (à cause de la restriction sur l'utilisation) qui peuvent amener à des pertes de capacités de calcul de l'ordre de 50% à 70% [And03]. De même, les ordonnanceurs de classe (3,3) sont les seuls à pouvoir garantir *en théorie* la ponctualité des tâches pour un taux d'utilisation de 100 % : c'est ce qu'on appelle l'*optimalité temps-réel*.

Remarque Si un ordonnanceur n'est pas optimal cela ne veut pas dire pour autant qu'il n'arrivera pas à trouver d'ordonnancement valide pour un ensemble faisable de tâches, mais cela veut dire qu'il y arrivera sous certaines conditions suffisantes (pas toujours nécessaires) qui restreignent les ensembles de tâches utilisables : on parle de conditions de *faisabilité*. Le paramètre classiquement utilisé est le *taux d'utilisation* des ensembles de tâches, c'est à dire la charge CPU nécessaire. Un ordonnanceur optimal garantit *en théorie* le respect de la ponctualité pour un taux d'utilisation égal à la capacité processeurs ($U = M$), et strictement inférieur à la capacité dans le cas non-optimal.

Cette contrainte liée au taux d'utilisation peut parfois être reportée sur d'autres paramètres : on parle des techniques par *augmentation de ressources*. Par exemple : *Global-EDF* n'est pas optimal, cependant Phillips et al. [PSTW97] ont montré que l'on peut considérer tout ensemble faisable de tâches s'il existe des processeurs $2 - 1/M$ fois plus rapides, M étant le nombre de processeurs initialement envisagé. Dans ce cas, l'augmentation de ressources est liée à la vitesse de fonctionnement des processeurs.

La majorité d'entre eux présentent des problèmes d'extensibilité comme noté par Brandenburg et al. [BCA08] dans leur portage de certains ordonnanceurs sur un noyau linux temps-réel (LITMUS) amenant même les ordonnanceurs partitionnés (P-EDF) à de meilleurs résultats dans certains cas. L'ordonnancement préemptif des traitements ne trouve son intérêt que si le nombre de changements de contexte ne croit pas démesurément par rapport au nombre de tâches ou de cœurs de calcul disponibles. Quelques constats sont établis :

- les ordonnanceurs prennent un temps de calcul en temps linéaire pour la gestion des tâches (dépendants du nombre de tâches) : à chaque appel à l'ordonnanceur, un temps précieux est passé à organiser la liste des tâches actives ;
- les stratégies d'allocation engendrent globalement un grand nombre de préemptions et

migrations : celles-ci entraînent un changement de contexte qui prend un temps non nul pour s'effectuer ;

- on voit apparaître des problèmes de contention au niveau de l'accès à la mémoire par le bus : si les changements de contexte se font simultanément sur plusieurs processeurs (comme c'est initialement le cas pour la famille Pfair [BCPV93]).

Les ordonnanceurs non-optimaux (e.g. Global-EDF, tout algorithme à priorités fixes) engendrent généralement moins de préemptions et migrations de tâches que les optimaux. Cependant, ils ne garantissent pas le respect des contraintes temporelles au dessus d'une borne supérieure du taux d'utilisation [CFH+04], ou alors en utilisant une technique par augmentation de ressource. Il est donc nécessaire d'utiliser plus de ressources pour obtenir des ordonnancements corrects pour le même ensemble de tâches. De plus, des conditions nécessaires et suffisantes pour prouver l'ordonnançabilité n'existent pas toujours. Enfin, les temps de repos des processeurs ne peuvent pas toujours être systématiquement évités, ainsi de précieuses ressources sont perdues.

Les ordonnancements en-ligne optimaux atteignent une limite d'utilisation en théorie égale à la capacité totale du système, mais sont souvent critiqués sur leur complexité et sur le fait qu'ils produisent beaucoup de préemptions et de migrations de tâches [Sri03]. Par conséquent, ils sont en théorie efficaces mais ne sont pas nécessairement utilisables en pratique en prenant compte les coûts de préemptions et de migrations : ce coût correspond aux changements de contexte et également aux décisions d'ordonnancement.

L'ordonnanceur doit être lui-même performant : l'optimalité est la garantie de pouvoir exploiter la pleine capacité des processeurs. Cependant en terme de complexité de calcul, l'exécution doit être *scalable* : si le nombre de tâches devient important, il faut que l'algorithme soit rapide pour éviter des surcoûts qui pourraient remettre en cause la ponctualité des traitements. Comment disposer d'un ordonnanceur optimal et rapide dans sa gestion des tâches ?

Dans un contexte multitâches, l'optimalité ne peut être atteinte que si les tâches peuvent être préemptées ou migrées. Comment limiter le nombre de changements de contexte engendrés ? L'algorithme d'ordonnancement doit pouvoir en limiter le nombre pour ne pas engendrer des surcoûts trop importants.

Nous proposons donc au chapitre 5 une approche pour mettre en place un ordonnancement efficace tout en maintenant des garanties théoriques fortes (optimalité). Cette nouvelle approche aura pour objectif de diminuer les préemptions et les migrations de tâches dans les ordonnancements temps-réel optimaux pour multiprocesseurs tout en ayant une complexité en-ligne faible. Notre méthode se décompose en deux étapes, l'une hors-ligne pour placer les travaux sur les intervalles, l'autre en-ligne pour les exécuter dynamiquement à l'intérieur de chaque intervalle.

2.4 Modèles de travail

Nous allons tout d'abord préciser les hypothèses de travail concernant le modèle de l'architecture, le modèle d'exécution et le modèle de tâches. Dans cette thèse, nous ne nous intéresserons pas à définir une architecture réseau, un nouveau modèle de tâche ou un nouveau modèle d'exécution mais plutôt à exprimer les propriétés attendues pour appliquer notre approche.

2.4.1 Description du modèle de l'architecture

L'architecture que nous considérons est une architecture parallèle composée de nœud à base de composants COTS.
1er niveau - le **réseau**, média de communication entre des noeuds.

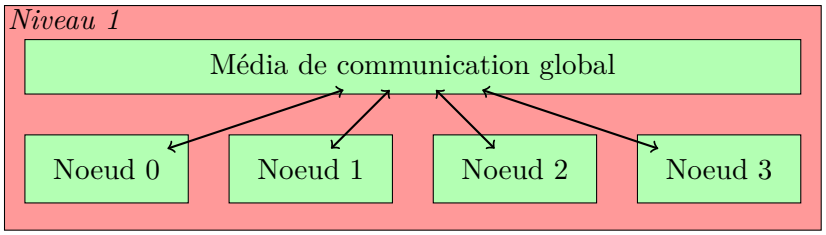


FIGURE 2.5 – Le modèle d'architecture au niveau global.

Chaque nœud peut accéder aux autres par l'intermédiaire du *média de communication global*. La gestion des communications à travers ce réseau dépend de sa topologie, des caractéristiques du réseau (bande passante, latence) et de son accès (Media Access Control - MAC). Les propriétés de l'architecture sont de plusieurs ordres :

1. la garantie des communications en temps borné et connu, i.e. il existe un majorant du temps d'acheminement des messages entre les nœuds,
2. un protocole d'accès permettant un partage garanti, i.e. l'accès au réseau pour l'émission et la réception de chaque nœud est assurée.

2ème niveau - le **noeud**.

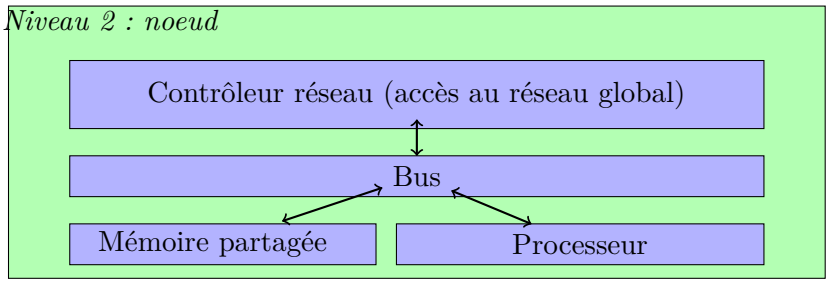


FIGURE 2.6 – Le modèle d'architecture au niveau d'un nœud.

Chaque nœud est composé d'une mémoire locale, d'un processeur disposant d'un ou de plusieurs cœurs de calcul accédant à la mémoire via un bus et communicant aux autres nœuds par l'intermédiaire d'un contrôleur réseau. Nous ferons l'hypothèse que les cœurs de calcul sont homogènes : leur fréquence horloge est identique et leur temps d'accès mémoire est similaire.

Justification du modèle de l'architecture

Afin d'évaluer la pertinence de nos hypothèses de travail, nous allons détailler quelques réseaux dans les systèmes distribués destinés au contexte embarqué pour des applications critiques.

La garantie sur le délai de transmission impose l'absence de collisions. Soit la topologie le permet comme les réseaux commutés, étoile ou point à point : les collisions peuvent alors être évitées avec des communications dirigées par les évènements (*event-triggered*). Soit il s'agit d'un medium à accès partagé et le protocole d'accès doit arbitrer les communications pour éviter les collisions soit par priorité comme le CAN (Controller Area Network), soit par découpage de plages temporelles (Time Division Multiple Access - TDMA) comme l'approche TTP [KG94] dont Kopetz et al. sont les précurseurs. L'une des constructions les plus simples consiste à attribuer périodiquement un slot de communication à chacun des nœuds. Une autre approche [JCD11], utilisée dans OASIS-D, consiste à exprimer précisément les contraintes de communications sous forme d'automates temporels et d'utiliser un système linéaire d'inégalités pour vérifier la ponctualité et obtenir un dimensionnement des slots propre à chaque nœud.

Au niveau de l'extensibilité, les topologies point à point présentent rapidement des problèmes d'encombrement, de poids et qui augmentent le coût de câblage. Dans l'automobile, on voit apparaître des approches TDMA avec le protocole Flexray [Ber01]. Les topologies proposées vont du bus à l'étoile par l'intermédiaire de commutateur, en exploitant des communications dirigées par le temps (*time-triggered*) et statiques [5] par des tables de communications embarquées sur chacun de nœuds. L'avantage principal est de savoir quel nœud doit émettre à quel moment, facilitant la vérification et la validation de tels systèmes. Ces réseaux vont au delà des propriétés énoncés car ils proposent des réseaux prédictibles et déterministes. Néanmoins, l'extensibilité du bus n'est pas parfaite : plus le nombre de nœuds est grand, plus la quantité d'information émise par nœuds est réduite.

Dans l'aéronautique, la topologie bus a été employée par le passé chez Boeing (avec SAFEbus [ARI93]), mais avec l'appui de composants dédiés. L'AFDX [ARI02] est également un réseau industriel utilisé chez Airbus avec l'A380, de type réseau Ethernet commuté donc à base de composant COTS. La maîtrise de la charge réseau s'appuie sur les commutateurs qui filtrent par contrat tout dépassement de bande passante venant des équipements reliés directement à eux. Il est ainsi possible d'utiliser des communications *event-triggered* tout en permettant une grande extensibilité. Des difficultés de l'analyse des délais de bout en bout existent, car les méthodes actuelles aboutissent notamment à des évaluations pessimistes [CSEF06]. Enfin, ce délai va dépendre de la charge réseau, ce qui complique l'analyse de propriétés temps-réel.

Nous nous intéresserons uniquement à utiliser un modèle réseau tel que défini par les deux propriétés énoncées plus tôt et ainsi de pouvoir proposer des résultats qui peuvent s'appliquer (ou s'adapter) à un certain nombre de réseaux embarqués temps-réel. Le modèle du nœud décrit ci-dessus en Fig. 2.6 peut s'adapter aux architectures MPSoC classiques.

2.4.2 Description du modèle d'exécution et justification

Nous faisons l'hypothèse que chaque nœud dispose de son propre OS temps-réel (*RTOS*). Ensuite nous supposons que l'allocation des tâches est partitionnée sur les nœuds, i.e. que les tâches sont réparties par groupe sur les différents nœuds. Les tâches peuvent être préemptées sur les cœurs par des tâches du même groupe. Les migrations sont autorisées entre cœurs d'un même processeur, donc d'un même nœud. En cas de reconfiguration une réallocation des tâches et des répliques peut avoir lieu, nous envisageons donc d'autoriser les migrations entre les nœuds dans ce cas précis.

5. même si une partie du protocole de Flexray autorise un slot dynamique mais ces communications ne sont pas contrôlées.

Les possibilités d'allocation de tâches devront être connues (déterminées) et prévisibles. De même, une connaissance à priori des caractéristiques temporelles et logiques des tâches est nécessaire.
Enfin, un *dimensionnement* doit être proposé : il correspond à l'analyse du comportement du système suivant différents critères permettant d'obtenir des majorants sûrs du système en fonctionnement.

2.4.3 Description du modèle de tâches

Nous considérons l'exécution de tâches critiques, i.e. qui sont essentielles pour le fonctionnement du système. Nous ferons l'hypothèse qu'elles sont toutes connues à priori et que leur comportement temporel est parfaitement établi :

- toutes les contraintes temporelles sont explicitement décrites. Ces tâches devront toujours être exécutées pour le bon fonctionnement du système. Ceci permet d'établir un système dont le comportement est *prévisible* et *reproductible* :
  - prévisible : soit t la date courante, le comportement du système est connu à l'instant $t+1$;
  - reproductible : pour un ensemble d'évènements E en entrée du système, le comportement du système restera identique.
- nous disposons de majorants sûrs de l'exécution des tâches (temps d'exécution, besoins en communication).

Dans la majorité des systèmes temps-réel, les tâches sont en grande partie récurrentes c'est à dire qu'elles sont exécutées régulièrement, il donc est nécessaire de gérer les tâches périodiques. Comme décrit plus tôt, l'expression de plusieurs comportements différents permet de décrire des modes correspondant à des phases de fonctionnement du système. Ces comportements peuvent s'exprimer par des temps d'exécution, des échéances et des lois d'arrivée différentes. Enfin, il est intéressant de pouvoir exprimer des branchements conditionnels, c'est à dire des comportements exclusifs fonction d'une condition par exemple. Nous nous intéresserons à utiliser un modèle de tâches disposant de comportements multiples et récurrents.

Justification du modèle de tâches

Les travaux théoriques concernent dans une majorité des tâches périodiques ou sporadiques, dont le calcul du taux d'utilisation est simple (rapport temps d'exécution sur période/échéance). Ce besoin d'expression se traduit actuellement sur le plan académique par des études récentes intégrant des modèles de plus en plus complexes (e.g. le modèle multiframe [BCGM99], récurrent [Bar98, Bar03], non-cyclique) disposant de tests de faisabilité polynomiaux, mais pas forcément pour des architectures multiprocesseurs. Ci-dessous dans la table 2.2 figure une synthèse des caractéristiques des modèles de tâches étudiés. Nous détaillons certaines de leurs caractéristiques :

1. leur nature i.e. si leur comportement est dirigé par le temps (TT) ou par les évènements (ET),
2. leur faisabilité mono ou multiprocesseurs, i.e. s'il existe un test d'ordonnançabilité mono ou multiprocesseurs,

3. la présence de comportements multiples comme l'expression de temps d'exécution différents, l'activation non constante, etc.,
4. s'ils sont récurrents par la présence de cycles,
5. s'ils sont conditionnels i.e. intégrant des branchements conditionnels.

Modèle de tâches	E/TT	Faisabilité	Comportements multiples	récurrents	conditionnels
Apériodique	ET	MultiProc	Non	Non	Non
Arbitraire [MC70]	ET	UniProc	Non	Non	Non
Périodique [LL73][Mok83]	TT	MultiProc	Non	Oui	Non
Sporadique [Mok83][BMR90]	ET	MultiProc	Non	Oui	Non
DCT [HL92a]	ET	UniProc	Non	Oui	Non
Transaction périodique [TC94]	ET	MultiProc	Oui	Oui	Non
Multiframe [MC96][MC97]	ET	UniProc	Oui	Oui	Non
Récurrent [Bar98, Bar03]	ET	UniProc	Oui	Oui	Oui
GMF [BCGM99]	ET	UniProc	Oui	Oui	Non
Rate based [JG99]	ET	UniProc	Non	Oui	Non
Intra-sporadique [AS00]	ET	MultiProc	Non	Oui	Non
Automate temporel [LDAV10]	TT	MultiProc	Oui	Oui	Oui
Non-cyclique [MNLM10, Bar10]	ET	Monoproc	Oui	Oui	Oui

TABLE 2.2 – Liste d'un certain nombre de modèles de tâches triés par ordre chronologique d'étude. En grisé, les modèles à l'étude dans cette thèse.

Nous pouvons constater que les modèles dérivés du modèle sporadique ont souvent des tests d'ordonnançabilité monoprocesseur. Les automates temporels sont une classe un peu à part puisque nécessitant un ordonnancement cadencé par le temps : leur intérêt pratique a cependant été démontré avec le modèle de tâches OASIS [AD98].

La plupart des modèles de tâches présentés ici s'appliquent bien à un contexte multicœur à mémoire partagée. Les tâches devant communiquer entre nœuds différents doivent cependant être décrites telles que le délai d'acheminement des messages par le réseau soit pris en compte. Plusieurs modèles correspondent à nos besoins, notamment le modèle périodique appelé *transaction* [TC94]. Une transaction est une succession de traitements ou de messages s'exécutant de manière périodique, les traitements ou les messages ont des temps d'exécution différents.

Dans les chapitre 4 et 3, nous utiliserons principalement les modèles périodiques, multiframe et nous montrerons comment notre travail s'adapte à un modèle par transaction. Au chapitre 5, nous étudierons principalement un modèle périodique et nous verrons comment l'adapter à un modèle multiframe ou plus généralement aux automates temporels.

2.5 Conclusion

Dans ce chapitre, nous avons détaillé les problèmes généraux que posent l'étude et la conception de systèmes temps-réel distribués pour des applications critiques. Nous nous intéressons

plus particulièrement aux aspects liés à la flexibilité des systèmes distribués temps-réel pour la tolérance aux fautes et à la performance au niveau local d'un nœud disposant d'un multiprocesseurs à mémoire partagée.

Concernant la flexibilité, nous constatons que les approches existantes pour la tolérance aux fautes reposent souvent sur la réplication logicielle et proposent une allocation statique des tâches et de leurs répliques. Néanmoins, il est intéressant de permettre la réallocation des tâches lors de défaillances permanentes pour atteindre une allocation optimisée sur les ressources restantes. Pour mettre en place une telle approche plusieurs problèmes se posent : il faut savoir comment modéliser le problème de l'allocation des tâches et de leurs répliques, quelles méthodes efficaces existent pour trouver ces allocations et quels critères doivent être optimisés. Nous nous proposons de réallouer les tâches à chaque défaillance, cette reconfiguration doit être bien dimensionnée pour un système critique, et les actions à effectuer doivent être connues et garanties dans leur exécution. Nous répondons à ces questions aux chapitres 3 et 4.

Concernant les performances dans l'exécution des tâches temps-réel, les politiques optimales d'ordonnancement multiprocesseurs présentent théoriquement l'avantage de pouvoir utiliser toutes les capacités de calcul en respectant les échéances. En pratique, les ordonnancements non-optimaux peuvent être plus performants s'ils engendrent moins de surcoûts que les optimaux. Ces surcoûts correspondent à la gestion des tâches souvent effectuée en temps linéaire (ou plus), les politiques qui engendrent beaucoup de changements de contexte (préemptions et migrations locales) et enfin des contentions au niveau du bus si les changements de contexte sont simultanés. Nous nous intéressons à obtenir des ordonnancements optimaux et efficaces. Pour cela, nous nous proposons d'établir une approche générant des ordonnancements valides minimisant les préemptions et migrations locales. Nous nous intéresserons à disposer d'un ordonnanceur rapide et dynamique pour un système robuste. Nous répondons à cette problématique au chapitre 5.

Chapitre 3

Réallocation dynamique pour la tolérance aux fautes

Sommaire

Nous proposons de mettre en place une approche hors-ligne permettant une flexibilité en-ligne dans l'allocation des tâches et répliques. Cette reconfiguration a pour but d'améliorer la tolérance aux fautes dans les systèmes distribués temps-réel subissant des défaillances multiples et permanentes. À chaque défaillance, notre objectif sera d'atteindre une allocation de référence, optimisée pour l'ensemble restant des ressources sur le système.

Nous faisons l'hypothèse que la mémoire embarquée est faible et que le stockage de l'ensemble des tâches sur chaque nœud n'est pas possible. Chaque reconfiguration doit être sûre pour un système critique. La reconfiguration est sûre si les différents mécanismes impliqués sont correctement dimensionnés et adaptés. Quelles actions doivent être entreprises pour atteindre une allocation de référence ? Nous présentons les mécanismes nécessaires : la reprise de l'activité par une réplique pour garantir la continuité d'exécution, la réplication et des migrations de tâches afin d'atteindre l'allocation de référence.

Nous modélisons les spécifications des allocations des tâches et de leurs répliques et montrons comment intégrer les ressources nécessaires aux mécanismes de la reconfiguration. Nous proposons de le faire dans un contexte général à l'aide d'un langage de programmation linéaire. Cette formulation est-elle efficace ? Celle-ci conduit souvent à la résolution de problèmes NP-difficiles, nous évoquons donc les alternatives possibles. Elle permet néanmoins de présenter les critères utiles permettant d'obtenir une allocation optimisée dans notre contexte.

Ces reconfigurations doivent être effectuées en temps borné afin de garantir une maîtrise du temps reconfiguration, utile par exemple pour l'évaluation de la fiabilité globale du système.

Nous montrons une technique permettant de trouver le nombre minimum de migrations à appliquer pour chaque reconfiguration envisagée. Pour s'assurer que tous les scénarios de défaillances sont pris en compte, nous faisons l'hypothèse d'une architecture symétrique et qu'ainsi nous pouvons alors traiter le problème par niveau de défaillances. Nous montrons comment explorer tous les scénarios dûs à un nombre fixé de défaillances. Ceci nous permet d'établir une politique de gestion de défaillances définie hors-ligne.

La conception de systèmes distribués avec des tâches critiques nécessite de pouvoir garantir leur exécution en présence de fautes, on parle de systèmes tolérants aux fautes. La technique la plus répandue pour tolérer les défaillances (issues de ces fautes) est la réplication. Le critère de défaillance unique a été communément étudié par le passé. Cependant, ce n'est pas suffisant pour un système qui requiert une haute sûreté de fonctionnement, notamment en présence de composants COTS. En général, la conception de tels systèmes se fait sur l'hypothèse que les tâches et leurs répliques sont allouées une fois pour toute.

3.1 Problématiques

Ce travail peut être vu comme une partie d'un processus FDIR (*Fault Detection followed by Isolation and system Reconfiguration*) dans lequel les phases de détection, de diagnostic et d'isolation sont déjà effectuées. De par la nature distribuée du système, ces premières phases nécessitent l'utilisation d'un consensus. La phase suivante de recouvrement dans laquelle notre méthode peut s'appliquer, exploite des données de reconfiguration calculées hors-ligne.

Nous proposons tout d'abord d'obtenir statiquement des allocations de références, c'est à dire un placement optimisé des tâches, utilisant efficacement l'ensemble des ressources disponibles. Ce placement optimisé peut être obtenu par un grand nombre de frameworks existants [IPEP05, JKH05, LKY+00, SJH+10]. Le critère d'optimisation dans notre contexte peut être multiple : par exemple équilibrer la charge CPU/réseau tout en maximisant le nombre de répliques allouées par tâche. Nous envisageons ensuite les allocations optimisées pour les sous-ensembles des ressources du système, suite à 1, 2, ..., D défaillances matérielles. Nous considérons que les ressources sont équitablement réparties sur les nœuds. Ainsi, nous envisageons successivement des défaillances matérielles permanentes amenant à la perte d'un nœud et par conséquent la perte des ressources, tâches et répliques allouées sur celui-ci. Ces allocations optimisées pour chaque niveau de défaillances sont appelées *configurations de référence*.

Lors d'une défaillance, les tâches perdues sont immédiatement reprises par une de leurs répliques via un mécanisme de basculement (*swact : switch of activity*). Le système passe d'une configuration de référence à une configuration quelconque suite à une défaillance. Le placement obtenu suite à défaillance peut ne pas être optimal vis à vis des critères recherchés (équilibre de la charge, nombre de répliques, ...) : nous la nommons *configuration dégradée*. Nous proposons alors de permettre les migrations et réplications nécessaires pour atteindre la configuration de référence pour le niveau de défaillance atteint, et ceci à chaque fois qu'une défaillance se produit.

Pour résumer notre méthode, elle consiste à :

1. déterminer les allocations de référence pour chaque niveau de défaillance (correspondant à un certain nombre de nœuds valides restants) ;
2. déterminer toutes les configuration dégradées, i.e. les allocations suite à tous les scénarios possibles de défaillances ;

3. pour chacune d'entre elle :
    - trouver le nombre minimal de migrations et de réplications nécessaires pour atteindre l'allocation de référence correspondante ;
    - déterminer l'ordonnancement des migrations et des réplications.
4. stocker les résultats de réallocation.

Dans la suite, nous allons détailler comment ces différentes étapes sont réalisées. Nous allons spécifier quels sont les critères pour la détermination des configurations de référence. Les configurations dégradées sont ensuite déduites par l'exploration de l'arbre de défaillance jusqu'à ce que le niveau maximal de défaillances envisagé soit atteint. La comparaison de l'allocation des tâches dans une configuration dégradée et dans une configuration de référence nous permettra de trouver l'ensemble de migrations nécessaires pour atteindre la configuration de référence. Une méthode efficace pour trouver le nombre optimal de migrations est le couplage par permutation des nœuds, car l'architecture considérée est homogène (ressources identiques sur chacun des nœuds). Enfin, il reste à trouver l'ordre des migrations respectant la consommation des ressources du système (mémoire, charge CPU) : plusieurs heuristiques sont envisageables pour y parvenir. L'ordonnancement nous permettra d'obtenir une évaluation sur la durée maximale de reconfiguration pour chaque niveau de défaillances : cet élément est nécessaire pour l'étude de la fiabilité globale du système. La fiabilité est l'attribut évaluant le fonctionnement correct du système, c'est à dire fidèle à ses spécifications.

Nous allons maintenant décrire les principales hypothèses nécessaires à la gestion des défaillances afin d'appliquer notre méthode.

3.1.1 Modèle de fautes

Les tâches sont redondées logiciellement, chacune d'entre elles a un nombre de répliques correspondant à l'objectif de tolérance aux fautes du système. Une réplique passive effectue des sauvegardes régulières de l'avancement de la tâche active correspondante (par exemple via un mécanisme de *checkpoints*). Nous considérons qu'une réplique consomme γ fois les ressources de sa tâche, $\gamma \in]0, 1]$.

Modèle de fautes Nous nous concentrons uniquement sur les défaillances matérielles des nœuds. Les défaillances sont la conséquence de fautes indépendantes et nous considérons qu'elles sont non-simultannées c'est à dire qu'il existe un temps non nul entre deux défaillances (nous disposons par exemple d'un MTBF : *Mean Time Between Failures*). Chaque nœud fournit les mécanismes nécessaires pour assurer un comportement *fail-stop* ou *fail-silent*. Nous envisageons les défaillances permanentes ou assimilées comme telles (e.g. lors d'un (re)démarrage d'un nœud d'une durée de plusieurs minutes ou plus), ce qui veut dire que toutes les tâches actives et répliques sont perdues. Nous supposons que les fautes transitoires sont gérées par d'autres mécanismes *sans changer l'allocation des tâches sur les nœuds* (comme par exemple un ordonnancement tolérant aux fautes introduisant une redondance temporelle des traitements).

3.1.2 Mécanismes nécessaires pour les étapes de la reconfiguration en-ligne

Comme évoqué ci-dessus, un processus dit *FDIR* se décompose en plusieurs phases. Nous présentons brièvement les deux premières phases que nous considérons comme acquises, puis nous détaillerons davantage la phase de reconfiguration qui fait l'objet de la présente approche.

Les phases de détection et d'isolation dans un système à comportement *fail-stop* nécessitent généralement un simple protocole de **diagnostic distribué** pour assurer une vue cohérente du système. Cette vue indique que chaque nœud dispose des mêmes informations sur la "santé" des autres nœuds du système i.e. quels nœuds sont considérés comme fautifs et sains. Un consensus implicite est possible si chacun des nœuds a connaissance de l'ordonnancement des communications et que tout le monde reçoit ce que chacun envoie (e.g. protocole TDMA dans un réseau *broadcast*). Sinon, un protocole *déterministe* (i.e. dont l'accord est toujours trouvé en temps borné et connu) peut être utilisé pour détecter les nœuds fautifs, émettre les vues entre les différents nœuds, et décider de manière uniforme de la vue globale. Des protocoles efficaces existent pour ce modèle de fautes, et même pour des modèles plus généraux gérant des fautes asymétriques tel que celui présenté par Serafini et al. [SBS+11]. Dans cet article, un réseau dirigé par le temps est utilisé. Cependant, comme souligné par les auteurs, pour l'implémentation de tels mécanismes un réseau synchrone n'est pas nécessaire, dès lors que l'on dispose d'une détection d'erreur périodique et que les émetteurs sont correctement identifiés. Le nombre de rounds (un round correspond à un envoi des vues pour chacun des nœuds) dépend du nombre de fautes consécutives durant la phase de diagnostic. Comme nous envisageons des défaillances non-simultanées, un nombre constant de rounds est suffisant ce qui veut dire que les phases de détection et d'isolation peuvent être réalisées en temps borné.

Notre travail se situe dans la phase suivant l'isolation. A ce moment là, certaines tâches et répliques sont perdues et ont été parfaitement identifiées. Pour appliquer la tolérance aux fautes nous supposons l'existence de plusieurs mécanismes. La continuité d'exécution des tâches impactées par la défaillance est réalisée par le biais de répliques passives dont le basculement d'activité est alors nécessaire lors de la perte de la tâche active. La réplication d'une tâche active est utile afin d'augmenter le niveau de tolérance aux fautes. Enfin, la reconfiguration peut demander une réallocation des tâches ou répliques, c'est à dire de migrer l'espace mémoire de tâches d'un nœud à l'autre de par la limitation de l'espace mémoire de chaque nœud.

Tout d'abord, l'allocation est faite de telle manière qu'une défaillance ne puisse entraîner la perte simultanée d'une tâche et de ses répliques donc les répliques et tâches actives respectives sont réparties. Ensuite, lors de la perte d'une tâche active, un mécanisme de basculement d'activité de réplique se déclenche (*swact*). A ce moment là, l'une des répliques devient la tâche active permettant de garantir la disponibilité : une augmentation de $1 - \gamma$ fois la ressource de la réplique est alors requise. De plus, nous autorisons les tâches (et répliques) à être réallouées pour atteindre une configuration de référence : cela nécessite un protocole de migration et de réplication. Le protocole de migration transfère le contexte et les données d'une tâche active d'un nœud source à un nœud cible. Le protocole de réplication utilise le même mécanisme de transfert mais sert à recréer une réplique passive distante de la tâche active. D'une manière générale, la reconfiguration en-ligne se déroule de manière séquentielle :

1. terminer l'exécution des répliques qui ne seront pas présentes sur les nœuds dans l'allocation à atteindre (celle de la configuration de référence) ;
2. migrer les tâches actives et répliques sur leur nœud cible ;
3. reprendre l'exécution des répliques concernées.

Les migrations des tâches actives et les réplications (le fait de créer une réplique passive distante) peuvent être réalisées par des mécanismes de migration pour systèmes asynchrones [MDP+00] dès lors que l'on peut borner leur temps de transfert ou l'adapter à un système temps-réel strict : c'est ce que nous verrons dans le chapitre 4. Après le transfert de la mémoire d'une tâche (pour

une migration ou réplication), celle-ci reprend son exécution. L'arrêt de l'exécution des tâches et répliques peut être nécessaire également dans le cas où le système se dégrade progressivement au delà du seuil de défaillance étudié, ou lorsqu'il ne reste pas assez de ressources sur un nœud pour effectuer un transfert (migration/réplication).

Délai total de reconfiguration Pour chaque scénario de défaillances, le délai total dépend en grande partie des temps de transferts (migration, réplication) : la détection de fautes et l'isolation étant en temps borné et généralement effectuées rapidement. Dans le cas de reconfiguration sans migration ou réplication, le délai total peut être considéré comme nul.

3.2 Positionnement

Comme indiqué dans l'état de l'art général, on voit apparaître des approches pour la gestion de la tolérance aux fautes dans certains contextes comme celui de l'aéronautique. Engel et al. [EJS+10] proposent de reconfigurer éventuellement aux différents démarrages du système en réallouant les applications aux modules (suivant la terminologie IMA) suite à la vérification des modules corrects et fautifs par un protocole d'accord. Le modèle de fautes considéré est celui des fautes byzantines. Le concept de *reconfiguration multi-statique* est employé pour signifier que toutes les configurations atteignables sont déterminées et certifiées hors-ligne, comme dans notre approche.

Dans le domaine des performances, des méthodes de calcul hors-ligne utilisent cette flexibilité en-ligne afin de minimiser la dégradation du temps d'exécution de tâches [LKP+10, SBEP11]. La méthode de Lee et al. [LKP+10] présente des similitudes par rapport à ce que nous proposons : la détermination statique d'allocations de références, la recherche de tous les scénarios de défaillances, la recherche du nombre optimal de migrations, le stockage des informations de reconfiguration. Cependant plusieurs différences apparaissent :

1. le modèle de l'architecture présente une mémoire partagée stable où les contextes et données des tâches sont sauvegardées : par conséquent aucune réplique n'est utilisée puisqu'une tâche perdue peut être reprise par un autre nœud en accédant à cette mémoire ;
2. les contraintes de ressources ne sont pas traitées que ce soit en terme de charge CPU, réseau ou d'empreinte mémoire : par conséquent aucun ordre dans les migrations n'est envisagé ;
3. la recherche du nombre optimal de migrations se fait par une méthode dont la complexité est exponentielle alors que nous proposons l'utilisation d'une méthode existante en temps polynomial.

Il existe peu de travaux de reconfiguration portant sur les stratégies d'allocation et la gestion de répliques pour la fiabilité et l'équilibre de la charge suite aux défaillances. Néanmoins, citons le travail de Lee et al. [HLH99]. Ils proposent deux stratégies de tolérance aux fautes dans un système distribué soumis à des défaillances de nœuds. La première consiste à allouer toujours une réplique par tâche. Lors d'une défaillance, la reprise des tâches actives perdues est possible grâce à leur réplique distante puis une deuxième étape consiste à "régénérer" les répliques perdues sur un autre nœud valide du système (ce que nous appelons réplication dans notre travail), afin de maintenir une tolérance à une défaillance. Leur deuxième stratégie est statique et consiste à disposer de plusieurs répliques par tâche. A chaque défaillance, les tâches perdues sont reprises par une de leurs répliques. Ainsi la tolérance aux fautes diminue progressivement. Ils comparent ces

deux stratégies par simulation à l'aide de chaînes de Markov pour évaluer la fiabilité du système. Leurs résultats montrent que leur technique par "régénération" obtient les meilleurs résultats. En comparaison, nous proposons de réallouer (au besoin) l'ensemble des tâches et répliques lors d'une défaillance pour atteindre une allocation optimisée alors que Lee et al. se limitent à la réallocation des répliques perdues (dans leur première stratégie) sans objectif particulier quant à la qualité de l'allocation.

Quelques frameworks existent pour des systèmes permettant une certaine forme de reconfiguration telle que [PCGI04] où les décisions sont prises en-ligne par l'analyse de réseaux de Pétri en fonction du nombre de défaillances détectées. Un module de décision choisit la meilleure stratégie en terme de fiabilité suite à cette analyse. Cependant dans cette approche, une allocation statique des composants sur l'architecture matérielle est considérée.

Des reconfigurations totalement dynamiques ont également été considérées et implémentées dans des systèmes Télécom [SPG03], mais au détriment d'un délai connu pour la gestion des défaillances.

3.3 Démarche proposée

Notre approche présente une méthode hors-ligne pour réallouer les tâches et leurs répliques en-ligne de manière sûre et en temps borné. Les étapes hors-ligne sont les suivantes : 1) exploiter les configurations de référence (allocations optimisées vis à vis de la charge et de la redondance) pour chaque niveau de défaillances considéré, 2) explorer un arbre de défaillances jusqu'à la profondeur voulue soit atteinte afin de trouver toutes les configurations dégradées (les allocations directement atteinte après défaillance), 3) comparer configurations de référence et dégradées pour chaque niveau de défaillance afin de trouver l'ensemble optimal de migrations/réplications de tâches nécessaire pour atteindre la configuration de référence.

Avant de décrire la manière d'exploiter une séquence de configurations de référence, nous allons exprimer formellement les différentes notions.

3.3.1 Modélisation

Afin de formaliser les concepts de notre méthode, nous allons exprimer les spécifications des configurations à l'aide d'un langage de programmation linéaire.

Une **configuration** correspond à une allocation des tâches et de leurs répliques sur les nœuds. Soit $\mathcal{C}$ une configuration, nous mettrons en exposant le nombre de nœuds considéré et nous indicerons par d si cette configuration est atteinte suite à une défaillance. $\mathcal{C}$ est l'ensemble des X_p tel que X_p est défini comme l'ensemble de tâches et répliques affectées au nœud p. Ci-dessous un exemple d'une architecture à 6 nœuds dont le placement initial est donné par $\mathcal{C}^6$ et suite à la défaillance du nœud 3 atteint la configuration dégradée $\mathcal{C}_d^5$:

$$\mathcal{C}^6 = \{X_1, X_2, X_3, X_4, X_5, X_6\}, \mathcal{C}_d^5 = \{X'_1, X'_2, X'_4, X'_5, X'_6\}$$

où X'_p correspond à l'allocation des tâches et répliques sur le nœud p après une défaillance et par conséquent après un basculement d'activité des répliques du nœud p éventuellement concernées.

Une *configuration de référence* correspond à une allocation satisfaisant une condition d'admissibilité et optimisant une fonction économique. La condition d'admissibilité et la fonction

économique peuvent être toutes les deux considérées comme des "boites noires" qui dépendent du contexte applicatif (voir Section 6.2.1).

A titre d'exemple, nous allons maintenant détailler les spécifications d'une configuration de référence suivant la terminologie de la programmation linéaire. Ce type de configuration doit respecter des contraintes d'admissibilité même si des défaillances se produisent. Ces contraintes d'admissibilité sont composées d'exigences liées à la tolérance aux fautes et aux ressources du système (empreinte mémoire, charge CPU et réseau qui diffèrent selon le domaine ciblé). Une configuration de référence présente trois propriétés :

1. elle permet une répartition équilibrée de la charge ;
2. elle respecte des contraintes de placement ;
3. elle respecte des contraintes de ressources.

Répartition équilibrée On peut par exemple utiliser un algorithme de placement par ordre décroissant des charges des tâches de telle manière à équilibrer la charge sur les différents nœuds (type LPT Largest Processing Time). On place les tâches puis leur réplique en allouant en priorité sur les nœuds les moins chargés.

Contraintes de placement Le placement doit permettre l'allocation de toutes les tâches et de leurs répliques. Nous notons $x^a_{t,p}$ la variable booléenne indiquant la présence de la tâche active t sur le nœud p et $x^b_{t,p}$ la présence de la réplique de la tâche active t sur le processeur (nœud) p. Nous exprimons tout d'abord que toutes les tâches actives doivent être placées :

$$\sum_p x^a_{t,p} = 1, \forall t \tag{3.1}$$

Soit R_t l'ensemble des répliques de la tâche t, alors chacune des répliques doit être allouée au plus une fois si les ressources le permettent. En effet, en cas de perte d'un nœud les ressources du systèmes diminuent et peuvent en pas être suffisantes pour l'exécution de l'ensemble des tâches et répliques. Nous exprimons la présence des répliques de la manière suivante :

$$\sum_p x^b_{t,p} \leq \sum_p x^a_{t,p}, \forall b \in R_t \tag{3.2}$$

Une tâche et ses répliques doivent être obligatoirement placées sur des nœuds différents :

$$x^a_{t,p} \leq 1 - x^b_{t,p}, \forall t, \forall p, \forall b \in R_t \tag{3.3}$$

Si on a plusieurs répliques, elle doivent elles-mêmes être placées sur des nœuds différents :

$$\sum_{b \in R_t} x^b_{t,p} \leq 1, \forall t, \forall p \tag{3.4}$$

Contraintes de ressources L'architecture matérielle est composée de M nœuds de capacité identique : nous notons $C_{p,r}$ la capacité du nœud p pour la ressource r. La ressource r peut représenter la capacité mémoire, CPU ou réseau. Chaque tâche t a besoin d'une quantité de ressource $w_{t,r}$ pour être exécutée sur un nœud, et ses répliques de $\gamma * w_{t,r}$. Une contrainte générale pour l'allocation des tâches sur chaque nœud peut être exprimée telle que :

$$\sum_t w_{t,r} x_{t,p} \leq C_{p,r}, \forall r, \forall p \tag{3.5}$$

Nous allons nous concentrer ici uniquement sur la ressource CPU et nous noterons sa capacité C_p. Nous allons voir en quoi l'utilisation d'une réplique passive et du *swact* modifient les contraintes de ressources.

Il faut d'abord que toutes les tâches actives puissent s'exécuter en respectant la capacité des nœuds. Soit w_t les ressources CPU consommées par la tâche active t (simple adaptation de l'Eq. 3.5 pour l'ensemble des nœuds) :

$$\sum_p \sum_t w_t x_t \leq M C_p \tag{3.6}$$

Cette condition n'est en fait pas suffisante car il faut pouvoir s'assurer de l'exécution des répliques et que toutes les tâches actives perdues lors d'une défaillance sont reprises par leur réplique répartie sur les nœuds restants. Comme nous construisons un système tolérant au fautes, nous devons vérifier que chaque configuration peut gérer une défaillance sans surcharge. Le mécanisme concerné est le *swact* (*switch of activity*) d'une réplique en tâche active. Nous allons (re)montrer (montré par Sirdey dans sa thèse [Sir03]) que la charge active sur les nœuds doit être inférieure à la capacité de ressources des nœuds pour la prise en compte du basculement d'activité, et dépend d'un taux de remplissage moyen.

Considérons le cas idéal où la répartition suite à l'allocation des tâches et de leur réplique sur les M nœuds est parfaitement équilibrée. Soit $\tau = \frac{c}{C_p}$ le taux de remplissage moyen des nœuds, avec c la capacité des ressources utilisées par les tâches actives et par nœud.

Par rapport à l'Eq. 3.6 il faut ajouter la consommation des répliques sur chaque nœud pour les tâches actives des autres nœuds du système : en tout $(M-1)\gamma c$. Cependant la charge des répliques est uniformément distribuée, donc pour chaque nœud la consommation des répliques n'est que de $\frac{(M-1)\gamma c}{M-1} = \gamma c$. Nous obtenons donc que la capacité dépend de la charge des tâches actives et de celles des répliques :

$$c + \gamma c = C_p \tag{3.7}$$

La tolérance lors d'une défaillance n'est pas encore prise en compte dans la charge, dans notre contexte elle correspond au basculement passif/actif des répliques concernées, il faut donc la rajouter. Lors d'une défaillance, une charge active c est perdue (perte de la charge active d'un nœud). Cette charge perdue est répliquée sur chacun des nœuds restant d'une proportion égale à $\frac{\gamma c}{M-1}$. Donc pour le basculement passif/actif, il faut apporter la charge complémentaire $\frac{(1-\gamma)c}{M-1}$ sur chacun des nœuds restants. L'équation 3.7 devient :

$$c + \gamma c + \frac{(1-\gamma)c}{M-1} = C_p \tag{3.8}$$

Le taux de remplissage est donc dépendant du nombre de nœuds M et du ratio de la charge passive γ :

$$\tau = \tau(\gamma) = \frac{c}{C_p} = \frac{M-1}{M(\gamma+1) - 2\gamma} \leq 1$$

L'équation de contraintes de ressource 3.6 devient alors :

$$\sum_p \sum_t w_t x^a_{t,p} \leq M \tau(\gamma) C_p \tag{3.9}$$

Cela traduit le fait que l'allocation des tâches actives ne doit pas dépasser un certain seuil, strictement inférieur (si $\gamma > 0$) à la capacité des processeurs pour pouvoir ensuite placer les

répliques et permettre l'application de la tolérance aux fautes (basculement de l'état de réplique à l'état de tâche active).

Finalement, nous voulons être capables de transférer les données nécessaires de chaque tâche sur chacun des nœuds pour la migration ou la réplication. Soit $w^m_{t,r}$ le coût de migration, défini par les ressources nécessaires pour garantir qu'une tâche t peut être transférée d'un nœud à l'autre, nous notons w^m_t si l'on ne considère qu'une seule ressource. Si plusieurs migrations doivent s'effectuer au départ ou à destination d'un nœud p, nous pouvons effectuer ces migrations de manière séquentielle afin de ne comptabiliser qu'un seul coût de migration à la fois. Nous pouvons exprimer la contrainte CPU pour la tâche ayant le plus grand coût de migration, afin de le prendre en compte dans les contraintes de ressources de chaque nœud. Basé sur l'Eq. 3.5 cela donne :

$$\sum_t w_t x_{t,p} + \max_t(w^m_t) \leq C_p, \forall p \tag{3.10}$$

Nous considérons que toute technique de migration peut être utilisée dès lors qu'elle garantit un traitement en temps borné. Pour plus de détails, nous verrons dans le chapitre 4 une manière précise de prendre en compte les migrations pour des tâches temps-réel strictes plus complexes.

Le principal objectif peut être par exemple de maximiser le nombre de tâches et de leurs répliques. Pour suivre notre modèle entier, nous définissons une fonction économique :

$$\text{Maximiser} \sum_p w_t x^a_{t,p} + \lambda \sum_p w_t x^b_{t,p} \tag{3.11}$$

où λ est un facteur constant défini par l'utilisateur, utilisé pour donner un poids aux répliques par rapport à l'allocation des tâches actives. Ce paramètre est donc nécessairement inférieur ou égal à 1.

En ce qui concerne le modèle présenté et la plupart des problèmes d'allocation de tâches, ils sont souvent NP-difficiles dûs à la présence habituelle de contraintes de type sac à dos. En considérant des contraintes telles que celles rencontrées pour l'ordonnançabilité et la tolérance aux fautes pour allouer des tâches, le problème est NP-difficile même pour une seule défaillance comme l'ont montré Oh et al. [OS97]. Ce modèle nous a néanmoins permis de définir formellement les spécifications des configurations de références.

Notons que pour des problèmes simples (e.g. équilibre de la charge sous contraintes de type sac à dos [SPG03]), plusieurs heuristiques de grande qualité existent, par exemple [Gra66, Gra69, KK82]. De plus, dans des configurations plus complexes (e.g. partitionnement de graphe sous contraintes de type sac à dos et de routage [SD08]), des techniques de conception avec des algorithmes génériques peuvent être utilisées telles que le recuit simulé [EB08] ou GRASP [RR03]. L'approche GRASP est particulièrement adaptée à la conception d'algorithmes stochastiques performants (combinant algorithmes par construction progressive randomisés et recherche locale) à partir d'heuristiques élémentaires de partitionnement.

Dans la section suivante, nous présentons une manière d'exploiter une séquence d'allocations afin de fournir une méthode efficace de reconfiguration pour la tolérance aux fautes.

3.3.2 Séquence d'allocations de référence pour chaque niveau de défaillance

Pour illustrer notre approche, nous utilisons un exemple initialement proposé par Emberson et al. [EB08] pour montrer la dégradation de l'*utilité* d'un système embarqué temps-réel utilisant une allocation statique des tâches et de leurs répliques. Rappelons que l'*utilité* est une métrique définie par Shelton [She03a] pour évaluer le niveau de service d'un système soumis à des défaillances, elle dénote l'importance de la fonction d'une tâche dans le système. L'exemple que nous présentons est celui d'un ensemble de quatre tâches périodiques indépendantes (A, B, C, D) comme définie dans la Table 3.1. Chaque tâche est *critique* et dispose d'un maximum de deux répliques actives ($\gamma = 1$) qui devront être exécutées sur des nœuds différents. Enfin, à chaque tâche est associée son *utilité*.

Tâches	WCET	Période	Répliques	Utilité
A	7	10	2	0.5
B	7	10	2	0.3
C	2	10	2	0.15
D	2	10	2	0.05

TABLE 3.1 – Ensemble de tâches périodiques avec leur utilité système.

Considérons une architecture à quatre nœuds ($M = 4$) dans laquelle l'ensemble de tâches présenté doit s'exécuter, chaque nœud dispose d'un processeur et la topologie réseau est supposée symétrique. Cette topologie symétrique permet de permuter les nœuds sans remettre en cause les tests d'admissibilité de l'allocation (e.g. un test d'ordonnançabilité). Les ressources du système sont initialement suffisantes pour allouer toutes les tâches actives avec toutes les répliques.

Nous nous proposons de voir quelles sont les configurations de référence pour chaque niveau de défaillance, i.e. sans défaillance, avec une puis deux défaillances : nous noterons $\mathcal{C}^6, \mathcal{C}^5, \mathcal{C}^4$ les configurations de référence correspondantes. Toute allocation intégrant A ou B avec C ou D sur les mêmes nœuds respecte une utilisation totale inférieure à la capacité processeur, voir l'exemple sur la Fig. 3.2. Dans ce premier exemple, *nous ne nous occupons pas de l'utilité*. La figure 3.2 présente l'évolution de l'allocation des tâches et leurs répliques suite à deux défaillances, avec à gauche un système statique utilisant uniquement le *swact* (basculement réplique/tâche active) et à droite un système autorisant en plus des migrations et réplications. Voici une des configuration de référence sans défaillance :

$$\mathcal{C}^6 = \{X_1, X_2, X_3, X_4, X_5, X_6\}$$

avec $X_1 = \{A, D\}$, $X_2 = \{A', D'\}$, $X_3 = \{A'', D''\}$, $X_4 = \{B, C\}$, $X_5 = \{B', C'\}$, $X_6 = \{B'', C''\}$, comme indiqué sur la Fig. 3.2. Toute permutation de l'allocation nœuds reste correcte, ce qui veut dire que nous disposons d'un ensemble d'allocations de référence.

Plus généralement, notons que le processus d'optimisation pour l'allocation peut aussi dépendre de plusieurs critères, par exemple la minimisation de la charge réseau, l'équilibre la charge CPU [SPG03].

Avec une défaillance, une configuration de référence $\mathcal{C}^5$ est une allocation qui permet d'exécuter toutes les tâches et au moins une de leur réplique. Avec deux défaillances, toute allocation présentant toutes les tâches avec une réplique peut être définie comme configuration de référence $\mathcal{C}^4$ si l'ensemble des tâches et répliques respectent une utilisation totale inférieure à la capacité processeur sur chacun des nœuds. Un exemple d'allocations de référence est présenté pour

chaque niveau de défaillance, comme montré sur la figure 3.1 en comparaison des configurations dégradées obtenues après une et deux défaillances.

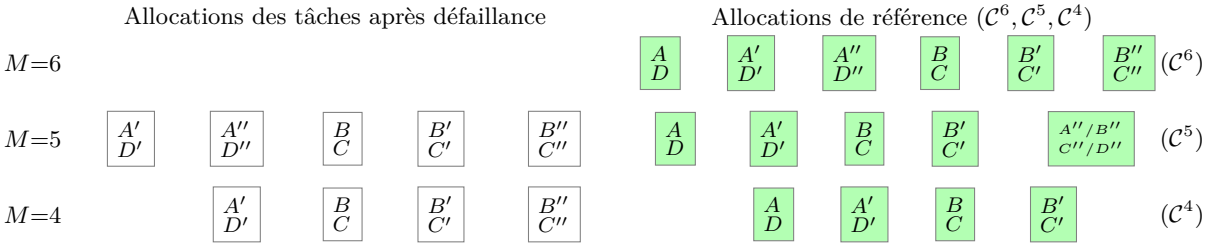


FIGURE 3.1 – Allocations des tâches après défaillances et leurs allocations de référence correspondantes. À l'état initial sans défaillance ($M = 6$), l'allocation des tâches est celle de référence. La notation A/B signifie "allocation de la tâche A ou B".

Comparons maintenant les reconfigurations possibles pour ces deux systèmes présentés sur la figure 3.2. Les défaillances envisagées sont celles du nœud 1 puis du nœud 2, il correspond initialement au pire cas de défaillances car entrainant la perte de deux répliques d'une même tâche (ici A et D). Au bout de la deuxième défaillance, la version statique du système se retrouve sans tolérance aux fautes. Lors d'une défaillance dans le système statique, si une réplique est perdue il n'y pas de changement envisagé, par contre en cas de perte d'une tâche active l'une des répliques prend le relai : c'est le cas dans l'exemple de la figure 3.2 où le nombre de répliques de A et D diminue progressivement.

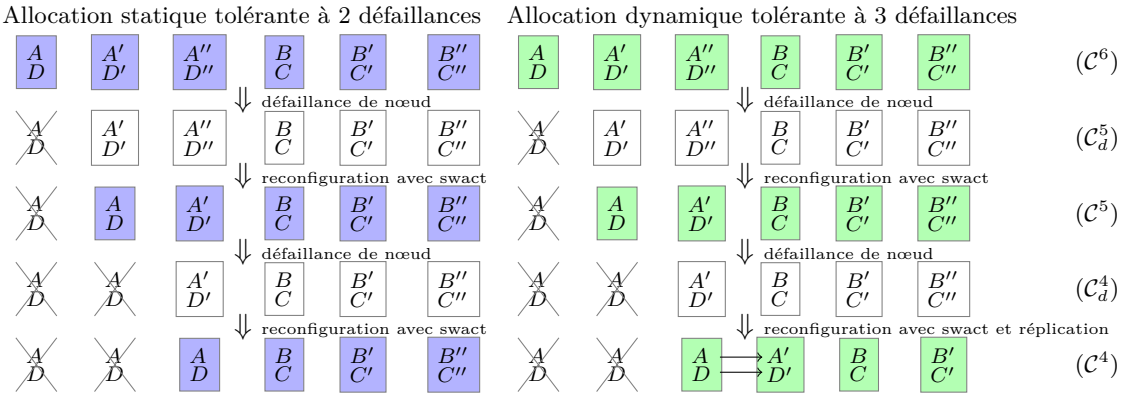


FIGURE 3.2 – Une architecture à six nœuds subissant deux défaillances : comparaison entre reconfiguration statique et dynamique.

Contrairement à l'approche statique, le système de droite permet la migration et la réplication de tâches entre les nœuds. Comme montré sur la figure, après deux défaillances, le système fournit toujours une tolérance aux fautes complète. Après la première défaillance, nous pouvons voir qu'aucune migration de tâche n'est nécessaire car après le basculement des répliques, une allocation de référence est atteinte. Cependant après la seconde défaillance, deux tâches peuvent être transférées (A et D) pour fournir de nouvelles répliques sur le nœud 3. Par conséquent, les tâches déjà allouées sur ce nœud (B et C) doivent être terminées pour libérer les ressources nécessaires à la réplication.

Il est aussi possible de fournir des allocations de référence si le nombre de défaillances dépasse le seuil prévu. Nous considérons maintenant le même ensemble de tâches avec leur *utilité* et avec uniquement quatre nœuds. Ici nous évaluons les allocations de références jusqu'à qu'il n'y ait plus de nœud. Dans le cas présent, le critère d'optimisation n'est pas le même puisque nous savons que suite à plusieurs défaillances toutes les tâches ne pourront plus s'exécuter. Nous utilisons ici l'utilité comme critère principal. Si nous devions utiliser le modèle linéaire présenté en section précédente, la fonction économique deviendrait :

$$\text{Maximiser} \sum_p U_t * x^a_{t,p} + \lambda \sum_p U_t * x^b_{t,p} \tag{3.12}$$

où U_t est l'utilité système d'une tâche t et λ le facteur évoqué précédemment.

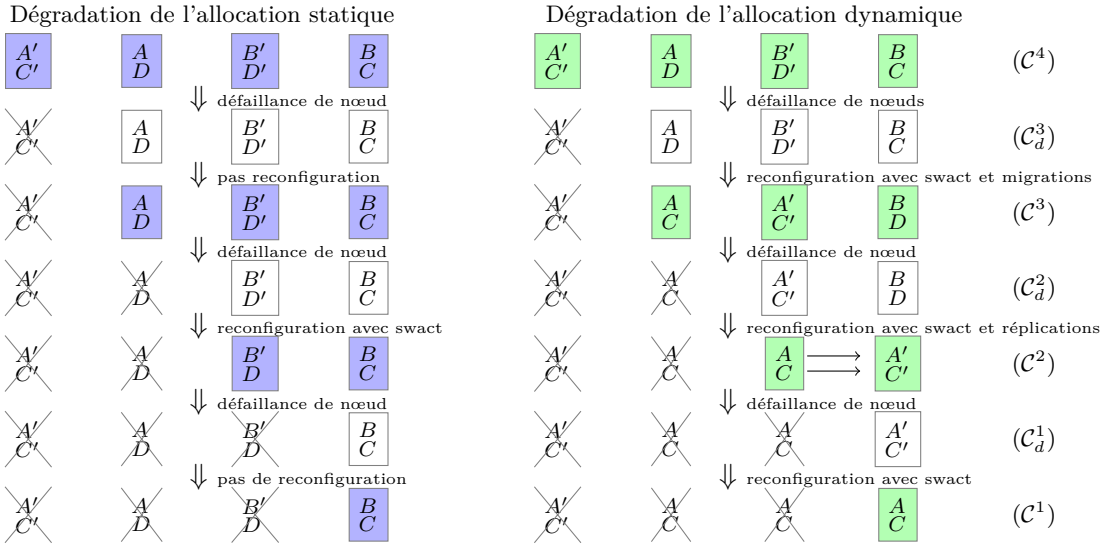


FIGURE 3.3 – Dégradation contrôlée de l'architecture à quatre nœuds avec reconfiguration statique et dynamique.

Pour l'allocation statique, le nombre de défaillances maximal envisagé va influencer le choix de l'allocation initiale, ce qui veut dire que l'allocation de tâche qui maximise l'utilité système pour deux défaillances n'est pas forcément celle qui maximise l'utilité pour plus de défaillances. Chaque allocation initiale différente peut présenter un profil différent en terme d'utilité (voir la figure 2.3, page 28). Comme remarqué par Emberson et al. [EB08], la meilleure allocation initiale va dépendre des exigences voulues en terme de tolérance aux fautes.

Contrairement aux approches statiques, la présente méthode permet de ne pas avoir de compromis à faire entre différentes allocations initiales : elle maintient une allocation de référence quelque soit le nombre de défaillances. La fig. 3.3 montre la dégradation de l'allocation des tâches du système dans une approche statique et dans une approche dynamique. Notre solution amène à garder les tâches A et C disponibles car ce sont elles qui maximisent l'utilité du système tout en respectant les conditions d'admissibilité.

Après la deuxième défaillance, les tâches A et C sont migrées du troisième au quatrième nœud, ce qui implique d'arrêter sur ce nœud les tâches B et D. Cette reconfiguration diminue

l'utilité du système mais amène vers une configuration de référence (plus appropriée en terme de tolérance aux fautes et d'utilité). Lorsque la troisième défaillance se produit (il reste seulement un nœud), l'utilité système atteint 0.45 ($U_B + U_C$) pour l'approche statique, comparé au 0.8 ($U_A + U_C$) dans notre cas. L'évolution de l'utilité du système est présentée en Fig. 3.4 en fonction du nombre de défaillances (défaillance du premier, second et troisième nœud) pour l'approche statique et pour l'approche dynamique de la Fig. 3.3.

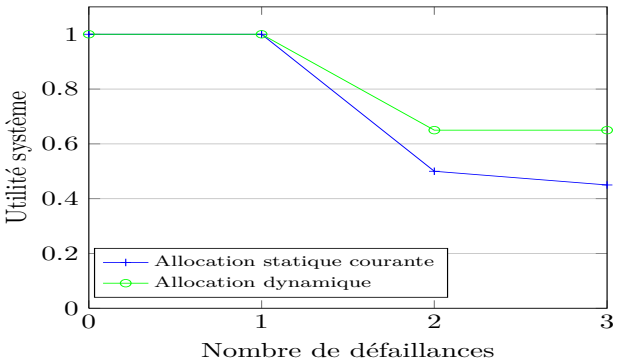


FIGURE 3.4 – Dégradation contrôlée : comparaison de l'utilité système en fonction du nombre de défaillances.

Nous avons vu dans cette section comment exploiter une séquence d'allocations et nous avons illustré sur un exemple l'efficacité de notre méthode par rapport à une approche statique prévue pour un niveau fixé de défaillances et avec la dégradation contrôlée du système.

3.3.3 Exploration de l'arbre de défaillances des nœuds

Pour garantir des transitions sûres et en temps borné après la défaillance d'un nœud pour atteindre une allocation équivalente à celle de référence, nous avons besoin d'identifier toutes les configurations dégradées. De telles configurations correspondent à l'état de l'allocation des tâches du système juste après la détection de la défaillance et le basculement des répliques concernées. Pour ce faire, nous explorons chaque niveau de défaillance comme un arbre de défaillance : nous partons de la configuration de référence et regardons toutes les combinaisons de la perte d'un nœud afin d'en déterminer les configurations dégradées. Cette opération est répétée jusqu'à ce que le niveau maximal de défaillance soit atteint ou jusqu'à ce qu'il n'y ait plus de ressources disponibles (plus de nœuds, ce dernier étant envisageable uniquement pour de petits systèmes).

Une liste optimale de migrations de tâches

Comme souligné dans la section 3.3.2, un ensemble de configurations de référence peuvent être (implicitement ou explicitement) défini pour chaque niveau de défaillance. L'architecture considérée dispose d'une topologie globale symétrique : ceci indique que les propriétés de l'état du système sont invariantes par permutation des nœuds et par permutation de deux tâches équivalentes en termes de ressources. Lorsque la reconfiguration s'applique, le nombre de nœuds disponibles a diminué, et par conséquent de processeurs. Pour libérer des ressources une première étape consiste à arrêter toutes les tâches/répliques qui ne seront pas présentes dans l'allocation de référence. Soit X_p d'une configuration $\mathcal{C}$ (respectivement $X_{p'}$ d'une configuration $\mathcal{C}'$) l'ensemble des tâches allouées au nœud p (respectivement p') après la défaillance d'un nœud (respectivement dans l'allocation de référence). Si p doit être dans le même état que p', $|X_p \setminus X_{p'}|$ **mouve-**

ments (migrations, réplications) sont nécessaires. Soit P l'ensemble des nœuds disponibles et $x_{pp'}$ l'association du processeur p avec le processeur p', une permutation optimale (en terme de mouvements) des nœuds est donné par le programme linéaire entier suivant :

$$\begin{cases} \text{Minimiser } \sum_{p\in P}\sum_{p'\in P}|X_p \setminus X_{p'}|x_{pp'}, & \\ \text{s. l. c.} & \\ \sum_{p\in P} x_{pp'} = 1 & \forall p' \in P, \\ \sum_{p'\in P} x_{pp'} = 1 & \forall p \in P, \\ x_{pp'} \in \{0,1\} & \forall p, p' \in P^2, \end{cases}$$

Sur la Fig. 3.5, nous montrons les mécanismes requis après la première défaillance de nœud, pour atteindre une allocation équivalente à celle de référence par permutation des nœuds (passage de $\mathcal{C}_d^3$ à $\mathcal{C}^3$). Si nous appliquons le programme linéaire à cet exemple, nous trouvons $x_{11} = 1$, $x_{22} = 1$ et $x_{33} = 1$ ce qui traduit le fait qu'aucune permutation n'est nécessaire. Cependant, deux migrations de tâches doivent être réalisées (C et D) après l'arrêt des répliques (B' et D') et les réplications de tâches (A et C) pour obtenir l'allocation de référence.

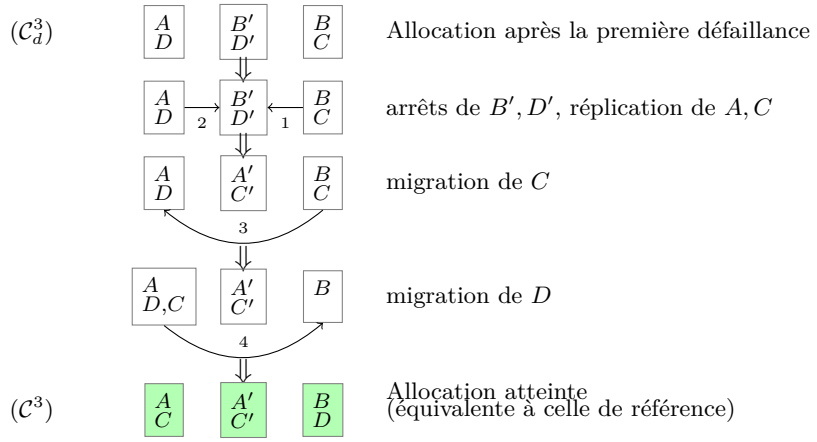


FIGURE 3.5 – Reconfiguration avec swact et migrations.

Ce programme est une instance du problème bien connu de matching sur un graphe biparti complet [1] qui est résolu en temps polynomial. Pour chacune des configurations dégradées, cette opération est répétée pour trouver la liste optimale de migrations.

Ordonnancement des migrations et délai de reconfiguration

Une fois que les ensembles de migrations de tâches ont été trouvés, il est possible d'évaluer le délai total de reconfiguration : si toutes les migrations sont séquentielles, cela correspond à la somme des délais de migrations. Les délais de migration dépendent de la technique utilisée pour le transfert des données, du contexte de la tâche et des caractéristiques du réseau (sa bande passante).

Soit $\mathcal{C}^n$ la configuration de référence dans laquelle le système fonctionne de manière nominale sans défaillance avec n nœuds et $\mathcal{C}_d^n$ la configuration dégradée après la perte du processeur $n+1$. Nous pouvons définir pour chaque nœud la durée de reconfiguration par le nombre de

1. Notons que lorsque la topologie est pas totalement symétrique, cette approche peut toujours s'appliquer dans une certaine mesure, même si le graphe biparti sous-jacent n'est pas complet.

migrations/réplications que il faut effectuer pour passer d'une configuration dégradée $\mathcal{C}_d^n$ à la configuration de référence de même nombre de nœuds $\mathcal{C}^n$. Après avoir obtenu le nombre minimum de migrations (par couplage) on obtient un tableau T associant chaque processeur de $\mathcal{C}_d^n$ à $\mathcal{C}^n$. On peut exprimer la durée des migrations pour le passage d'une configuration à l'autre :

$$\forall p, p' = T[p], \mathcal{D}^p_{\mathcal{C}_d^n,\mathcal{C}^n} = \sum_t \delta_t^A * |x_{t,p}^A - x_{t,p'}^A| + \sum_t \delta_t^B * |x_{t,p}^B - x_{t,p'}^B|$$

Avec δ_t^A (respectivement δ_t^B) le temps mis par une technique de migration pour migrer une tâche active (respectivement répliquer une tâche active) d'un processeur à l'autre.
La durée globale de reconfiguration correspond *au pire* à la somme des temps pour réaliser les migrations/réplications sur chaque processeur :

$$D^{max}_{\mathcal{C}_d^n,\mathcal{C}^n} \leq \sum_p (\mathcal{D}^p_{\mathcal{C}_d^n,\mathcal{C}^n})$$

Remarquons que pour transférer les données pour la migration ou la réplication, les techniques de migrations tâches asynchrones peuvent être utilisées dès lors que le traitement peut être borné dans le temps. Pour des systèmes temps-réel stricts, la migration d'une tâche active d'un nœud à l'autre demande de respecter certaines contraintes spécifiques : ceci fait l'objet du chapitre 4.

Comme montré sur la Fig. 3.5, il est nécessaire d'avoir un ordre statique de migration, notamment dans des systèmes embarqués où les ressources (mémoire, CPU) sont limitées. Les tâches A et B ne peuvent pas être allouées sur le même nœud au même moment sans dépasser la charge maximum CPU, ce qui n'est pas le cas pour les tâches C et D. Les migrations de tâche peuvent être réalisées pour atteindre l'allocation de référence. Néanmoins, la tâche C doit être répliquée. Un ordre doit être défini pour assurer une reconfiguration correcte : dans cet exemple, la réplication de C suivi de la migration de C ou l'opposé. D'une manière générale, l'ordonnancement des migrations a été largement étudié (voir [SCKN07]).

Si la migration ne peut être réalisée sans respect les contraintes d'admissibilité, par exemple lorsque deux tâches doivent être échangées entre deux même nœuds mais ne peuvent pas être exécutées en même temps sur le même nœud sans dépasser les capacités des ressources (e.g. les tâches A et B dans notre exemple). Plusieurs solutions sont possibles :

1. Basculer en mode "hors-ligne" : cela consiste à terminer toutes les tâches et répliques présentes sur le système afin de libérer toutes les ressources, pour ensuite réaliser les migrations de tâches nécessaires et redémarrer le système ;

2. Minimiser l'arrêt de tâches pour libérer uniquement les ressources nécessaires sur les nœuds source et cible des migrations de tâches ;

3. Permettre des migrations indirectes : une tâche migrante pourrait être transférée indirectement par des nœuds intermédiaires moins chargés.

Finalement, le délai de reconfiguration global peut être évalué pour tous les niveaux de défaillances (évaluation d'une borne supérieure). Ce délai permet de garantir que quelque soit le scénario de défaillances, une configuration de référence sera atteinte en temps borné et connu. Cette information est importante pour étudier la fiabilité globale du système en prenant en compte les taux de défaillances des composants. Ces solutions peuvent s'appliquer également si le délai de reconfiguration global est considéré trop grand, la libération de ressources supplémentaires permettraient l'exécution de certaines migrations/réplications en parallèle afin de le

diminuer.

Dans cette section, nous avons présenté les différentes étapes de notre méthode et illustré qu'elle permet d'obtenir une tolérance aux fautes efficace par rapport à une approche statique.

3.4 Conclusion

Nous avons proposé une méthode hors-ligne pour permettre une flexibilité en-ligne de l'allocation des tâches et de leurs répliques. Cette méthode tend à améliorer la tolérance aux fautes permanentes dans l'exécution de systèmes temps-réel distribués. Notre objectif est d'atteindre à chaque défaillance une allocation dite de référence optimisée pour les ressources restantes.

Pour de tels systèmes, nous devons nous assurer que ces reconfigurations sont sûres. Nous avons identifié les mécanismes nécessaires (réplication, swact, migration). Nous avons utilisé une formulation linéaire pour spécifier correctement les allocations en présence de ces mécanismes, i.e. en prenant compte les ressources nécessaires pour leurs exécutions. Cette formulation nous a aussi permis de présenter les critères d'optimisation via des fonctions économiques impliquant l'utilité système et présences de répliques. Néanmoins celle-ci conduit souvent à la résolution de problèmes NP-difficiles, nous avons présenté d'autres alternatives efficaces.

Nous avons défini hors-ligne une politique de gestion de défaillances, identifié une méthode permettant de trouver le nombre de migrations minimum à effectuer pour atteindre une allocation de référence. Nous avons montré comment explorer tous les scénarios possibles de défaillances. Cette exploration est facilitée par la nature symétrique du système qui permet d'envisager les reconfigurations par niveau de défaillances.

Notre méthode peut s'avérer utile par exemple dans un contexte temps-réel strict. Pour y parvenir certaines hypothèses doivent être présentées. Pour l'instant, nous avons illustré notre méthode sur un exemple simple de tâches périodiques non communicantes. Pour un système avec des tâches temps-réel, notons que l'ordonnançabilité peut être vérifié par ce type de contraintes de ressources avec des modèles de tâches simples comme le modèle périodique. Pour considérer des modèles de tâches plus complexes avec des contraintes de communications, il peut être nécessaire de s'appuyer sur des tests d'ordonnançabilité plus appropriés : par exemple des analyses basées sur le temps de réponses (voir page 18) ou sur la demande processeur (voir page 69). Plusieurs approches proposent de garantir l'ordonnancement des tâches et des messages, voir page 18 pour la description des approches existantes.

Pour les systèmes distribués à priorité fixe, le travail de Palencia et al. [PGH98] considère les applications logicielles comme des chaînes de tâches communicantes avec des offset statiques ou dynamiques. Chaque tâche ou message dispose de ses propres contraintes temporelles et d'exécution. Les relations de précédence existent entre les tâches par le biais de messages émis et reçus. Ces dépendances forment des graphes orientés acycliques appelés *transactions*. Dans ce modèle, le test d'ordonnançabilité consiste à vérifier que les temps de réponse pire cas pour toutes les tâches sont inférieurs ou égaux à leur échéance en présence des interférences des autres tâches. De plus, ce test d'ordonnançabilité présente l'avantage d'effectuer ce test nœud par nœud, et non globalement comme c'est généralement le cas. Ce test d'ordonnançabilité peut être directement utilisé comme test d'admissibilité dans notre méthode pour trouver les allocations de référence.

Afin de prendre en compte la charge induite par les migrations de tâches pour garantir leur

exécution, nous proposons l'utilisation d'une tâche additionnelle de transfert intégrée sur chaque nœud concerné (le nœud source et le nœud cible), ce travail fait l'objet du chapitre 4.

Chapitre 4

Stratégies de décision et techniques pour la migration

Sommaire

Nous avons vu dans le chapitre précédent une approche de reconfiguration apportant une flexibilité opérationnelle dans un système embarqué temps-réel critique pour améliorer sa tolérance aux fautes. Dans cette approche, nous avons notamment proposé une méthode minimisant la migration et la réplication des tâches critiques afin de reconfigurer vers une allocation de référence suite à des défaillances matérielles.

Dans le domaine des systèmes distribués asynchrones, un grand nombre de mécanismes de migration de tâches ont été proposés. Toutefois, aucun travail similaire n'existe lorsque ces systèmes sont soumis à des contraintes temps-réel strictes. Nous présentons quatre techniques : la Copie Directe, la *Prefetch Précopie*, la *Prefetch Postcopie* et la Copie Mixte qui exploitent la description statique du comportement temporel des tâches. Ces techniques s'effectuent en temps borné tout en respectant les contraintes temps-réel, notamment les échéances de migration. Pour chaque mécanisme proposé, nous décrivons les conditions de faisabilité qui peuvent être vérifiées hors-ligne pour un temps de migration quelconque. Nous montrons également comment intégrer ces techniques de migrations dans des tests de faisabilité. Nous comparons les performances des techniques proposées par simulation. Nos résultats montrent que le surcoût induit par ces techniques est bas.

Ces travaux ont fait l'objet de 2 publications : un article court et un poster à RTAS'11 WiP [MJDF11b] et un article long à RTNS'11 [MJDF11a].

4.1 Problématiques

La migration de tâches a été largement étudiée dans les dernières décennies pour les systèmes distribués asynchrones [MDP+00]. La principale préoccupation de ces mécanismes est de réduire le *temps de gel* d'une tâche, l'intervalle de temps durant lequel la tâche migrée ne peut poursuivre son exécution. Permettre un tel temps de gel dans un système temps-réel strict aboutirait à un système peu prédictible et dont certaines échéances seront dépassées.

Actuellement, les systèmes temps-réel critiques tels que les systèmes automobiles ou avioniques de type contrôle-commande sont souvent développés pour des architectures distribuées. La sûreté de fonctionnement a un rôle important : même en cas de défaillances matérielles comme l'arrêt d'un nœud et par conséquent la perte des tâches allouées sur celui-ci, les tâches critiques doivent pouvoir continuer leur exécution pour assurer la disponibilité. Lorsque des défaillances se produisent, l'une des stratégies les plus répandues est de disposer d'une capacité de réplication dynamique nécessitant la gestion d'un ensemble de répliques des tâches critiques pour tolérer au mieux la dégradation du système distribué temps-réel strict. La conception de mécanismes de migration appropriés pour un système temps-réel strict avec des propriétés de prédictabilité est alors nécessaire. L'un des prérequis pour la conception de tels mécanismes de migration est de s'assurer que l'ensemble du système reste ordonnançable pendant la migration de tâches, en complément des démonstrations d'ordonnançabilité classiques utilisées avant et après que le processus de migration soit effectué [EB07]. Est-il possible d'appliquer ces migrations sans manquer d'échéance, en temps borné et sans aucun temps de gel ?

Dans ce travail, nous montrons comment construire des mécanismes de migration adaptés au contexte de systèmes distribués temps-réel stricts. De tels systèmes nécessitent une description statique et logique de leur comportement temporel, i.e. les contraintes temps-réel associées à chaque tâche ainsi que leurs chemins possibles d'exécution sont connus. *Aucun temps de gel n'est donc nécessaire pour migrer des tâches temps-réel stricts*, de par la connaissance des inactivités de la tâche à migrer. Nous présentons quatre techniques de migration qui illustrent cette affirmation. De plus, cette description statique des chemins possibles d'exécution permet d'identifier les zones mémoires utiles (soit en lecture, soit en écriture, soit en exécution) pour chacun des travaux d'une tâche temps-réel stricte. Nous utilisons cette relation pour organiser efficacement les transferts de données sur le réseau, en fonction de différentes stratégies (par des politiques de *Prefetch Précopie* ou *Postcopie*) et en fonction des contraintes temporelles de chacun des travaux. Nous formalisons les différentes contraintes temporelles qui décrivent les conditions de faisabilité de la tâche à migrer pour une échéance quelconque, définie par l'utilisateur (pouvant être supérieure à la période de la tâche). Des tâches additionnelles spécifiques sont utilisées pour implémenter la politique de migration de tâches sur chacun de nœuds. Ces tâches sont intégrées à l'ensemble des tâches pour vérifier l'ordonnançabilité CPU et réseau du système. Nous discutons des résultats de faisabilité obtenus pour différentes échéances de migrations possibles en section 4.5.1. Finalement, la section 4.7 présente une évaluation du coût induit par ces techniques en termes de préemptions supplémentaires engendrées.

4.2 Positionnement

Différentes techniques de migration ont été proposées pour les systèmes distribués asynchrones [MDP+00]. La Copie Directe est la plus communément utilisée car simple à implémenter. Elle consiste à geler l'exécution de la tâche qui va être migrée sur le nœud source, à transférer intégralement son espace d'adressage et à reprendre l'exécution sur le nœud cible. Cependant, cette technique induit un important *temps de gel*.
Plusieurs stratégies alternatives ont donc été proposées pour réduire ce surcoût. Par exemple dans la stratégie de Précopie [TLC85], la quasi totalité de l'espace d'adressage de la tâche est transférée pendant l'exécution de celle-ci sur le nœud source. L'opération de transfert est répétée pour les nouvelles pages modifiées par la tâche en cours d'exécution qui a altérée certaines d'entre elles déjà transférées. L'algorithme continue à envoyer les pages modifiées jusqu'à ce nombre de pages soit suffisamment bas. La tâche est alors gelée et les pages restantes transférées. Le temps de gel est donc réduit. D'autres techniques existent comme le Transfert à la référence, Flushing ou Freeze Free [Rou96]. Cependant, ces techniques exhibent tout de même un temps de gel. En outre, elles laissent également des dépendances résiduelles soit sur le nœud source soit sur un serveur de fichiers servant d'intermédiaire. Ces méthodes amènent donc à une augmentation du temps d'exécution des tâches migrées à la reprise de leur exécution sur le nœud cible à cause de défauts de pages, comme l'espace d'adressage n'est pas complètement disponible sur le nœud cible. Finalement, la tolérance aux fautes se trouve diminuée lors d'une défaillance de nœuds sources (ou de serveurs de fichiers) car leurs défaillances impliqueraient la défaillance des tâches migrées.

Quelques travaux existent pour les domaines du temps-réel souple ou les applications hautes performances. Constantinou et al. [CSM+05] montrent l'influence d'une migration de *thread* sur les performances dans une architecture multicœurs en fonction de différents paramètres comme la fréquence de migration et des modes de préchargement des caches. Briao et al. [BaBW08] la migration de tâches temps-réel souples dans les MPSoC avec quelques stratégies d'allocation et leurs résultats montrent qu'il y a un compromis réalisable entre la consommation d'énergie et le taux d'échéances manquées selon l'algorithme utilisé. Concernant les applications temps-réel multimédia, Acquaviva et al. [AAP08] ont proposé des supports pour la migration de tâche basés sur des points de reprise permettant le réplication ou la recréation de tâches. Cependant dans tous ces travaux, les protocoles de migration de tâches utilisés sont similaire à la Copie Directe et ne ciblent pas les applications temps-réel strictes, hypothèse faite dans le cadre de ces travaux de thèse.

Proche de notre travail, on trouve les protocole de changement de mode temps-réel [NGA09]. Cependant, ils se concentrent sur la définition des tests d'ordonnançabilité pour assurer des transitions correctes entre ensemble de tâches mais la migration de tâche n'est pas considérée à cause de l'ordonnanceur oisif utilisé. Récemment, l'étude de migration de lignes de caches de tâches temps-réel strictes a été réalisée pour des systèmes embarqués multicœurs [KRSM09]. C'est le travail le plus proche du notre. Les auteurs fournissent des garanties hors-ligne sur l'impact du pire temps d'exécution de tâches.

4.3 Analyse du contexte et hypothèses

Nous considérons un ensemble Γ de N tâches temps-réel multiframe s'exécutant sur des nœuds identiques d'un système distribué. Les nœuds sont connectés par un réseau garantissant une latence bornée δ et une bande passante bp tel que un protocole d'accès permette un partage

garanti, i.e. l'accès au réseau pour l'émission et la réception de chaque nœud est assurée. Ces propriétés font partie des hypothèses de travail que nous avons énoncées et décrites au Chapitre 2, Section 2.4. Chaque nœud $\mathcal{N}_k$ dispose de son propre ordonnanceur préemptif exécutant Γ_k, un sous-ensemble de Γ, tel que l'ensemble des sous-ensembles des tâches correspond à ensemble de tâche total ($\bigcup_n \Gamma_n = \Gamma$) et tel qu'il n'y ait pas d'intersection entre ces ensembles ($\bigcap_k \Gamma_k = \emptyset$). Chaque tâche T se caractérise par un tuple $(T.E, T.d, T.p)$ où $T.E$, $T.d$, $T.p$ représentent un vecteur de tous les temps d'exécution au pire cas des travaux, l'échéance relative des travaux et le temps de séparation entre l'activation de deux travaux indépendants i.e. la période de la tâche. Toutes les tâches suivent un cycle d'exécution tout comme le modèle multiframe [BCGM99], on parlera de *cycle de la tâche* : celui-ci se définit comme un multiple de la période $N_j * T.p$ où N_j est le nombre de travaux de la tâche T. Nous faisons l'hypothèse que les effets de cache sont inclus dans le WCET de chaque travail et pour chaque tâche. De plus nous faisons l'hypothèse que le WCET peut varier d'un nœud à l'autre donc nous considérons que le WCET est défini comme la valeur maximum des valeurs de WCET sur les nœuds du système distribué où la tâche est susceptible d'être exécutée.

En utilisant un modèle de tâches cycliques, les intervalles de temps durant lesquels une tâche ne peut s'exécuter sont statiquement connus hors-ligne (entre $T.d$ et $T.p$ dans le pire des cas). Cette propriété est une règle générale pour l'élaboration de systèmes temps-réel critiques [AD98]. Il doit être en effet possible de démontrer, à des fins de sûreté de fonctionnement, le comportement de systèmes temps-réel critiques, tel que les délais de bout en bout. De telles démonstrations ne peuvent s'effectuer que si le comportement temporel des tâches est statiquement décrit.

De plus, on suppose la mémoire est partagée entre les tâches d'un même nœud et que chaque travail dispose de son propre espace d'adressage i.e. il est possible de les distinguer. De nouveau, grâce à la description statique du comportement des tâches, cette identification est faisable. On note $s(T)$ la taille totale de la tâche i.e. la mémoire utilisée pour son espace d'adressage, $s(j)$ la taille mémoire de l'un des travaux et on définit la taille mémoire d'une tâche correspond à la somme des tailles mémoires de ces travaux : $\sum_j s(j) = s(T)$. On suppose que chaque travail dispose de la même empreinte mémoire. Nous envisageons ici uniquement des travaux indépendants, i.e. les travaux ne peuvent modifier qu'un sous-ensemble de leur propre espace d'adressage $s(j)$.

Nous considérons que la migration de tâches s'opère entre un nœud source, qui dispose initialement de tout l'espace mémoire de la tâche à migrer, et un nœud de destination, qui est la cible de la migration et ne dispose d'aucune page mémoire en lien avec la tâche à migrer.

On considère une tâche spécifique s'exécutant sur chaque nœud lorsqu'une demande de migration se produit, elle se comporte donc comme une tâche apériodique. Cette tâche appelée tâche système envoie ou reçoit exclusivement l'espace d'adressage de la tâche en cours de migration, voir Fig. 4.1. Cette figure présente le comportement temporel d'une tâche multiframe migrante T_m et de la tâche additionnelle T_σ pour un transfert entre la fin du deuxième travail et le début du suivant. Le nœud source exécute une tâche système d'émission tandis que le nœud de destination exécute une tâche système de réception. Cette tâche notée T_σ délivre des travaux système tels que la somme des temps nécessaires aux transferts de l'intégralité de l'espace mémoire de la tâche à migrer soit égale à : $\sum j_\sigma.e * bp = s(T_m)$ où $s(T_m)$ est la taille de l'espace d'adressage de la tâche à migrer. Cette durée totale, i.e. $\sum j_\sigma.e$, est utilisée pour envoyer/recevoir des données du réseau lorsqu'une demande de migration se produit. Notons que

plusieurs travaux de la tâche système peuvent être nécessaires pour effectuer cette migration, selon la politique de migration employée et la quantité de mémoire à transférer.

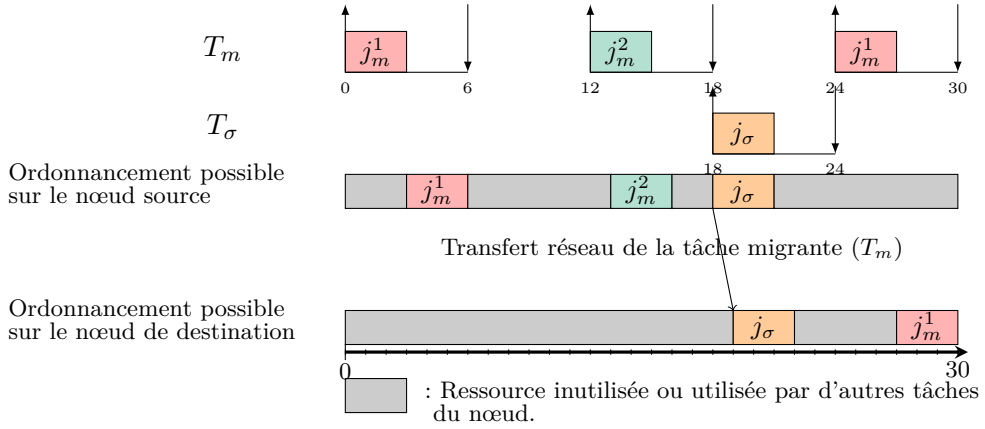


FIGURE 4.1 – Un exemple de tâche migrante T_m transférée par une tâche système composée d'un seul travail.

Nous envisageons uniquement la migration de tâches migrables, dont nous allons donner la définition. Pour simplifier, prenons l'exemple d'une architecture à deux nœuds telle qu'un ensemble de tâches Γ_S s'exécute sur le premier nœud $\mathcal{N}_S$ et telle qu'un ensemble de tâches Γ_D s'exécute sur le second nœud $\mathcal{N}_D$. On suppose qu'il n'y a pas d'intersection entre ces ensembles : $\Gamma_S \cap \Gamma_D = \emptyset$. Une tâche $T_m \in \Gamma_S$ est *migrable* de $\mathcal{N}_S$ à $\mathcal{N}_D$ ssi Γ'_S et Γ'_D sont deux ensembles ordonnançables sur leur nœuds respectifs $\mathcal{N}_S$ et $\mathcal{N}_D$, tels que le premier ensemble reste ordonnançable sans la tâche migrable et le second avec celle-ci : $\Gamma'_S = \Gamma_S - \{T_m\}$ et $\Gamma'_D = \Gamma_D + \{T_m\}$. Pour vérifier cette propriété, il peut être nécessaire d'utiliser un protocole de changement de modes temps-réel en considérant deux modes : pour le nœud source un mode avec l'ensemble de tâches initial et un mode sans la tâche à migrer T_m, pour le nœud de destination un mode avec l'ensemble de tâche initial et un mode avec la nouvelle tâche migrée T_m. Dans un système à priorité fixe, le changement de mode peut induire un changement de priorité des tâches des nœuds concernés et si le réseau n'est pas synchrone (horloge unique), il faut pouvoir garantir la transition entre les tâches se terminant dans l'ancien mode et le début de l'exécution des tâches dans le nouveau. Pour un comparatif sur ces techniques, le lecteur peut se référer à [RC04].

Dans la suite de ces travaux, nous supposons que cette propriété est vérifiée pour toute tâche à migrer, c'est à dire dont l'exécution sur un nœud ne remet pas en cause l'ordonnançabilité de l'ensemble des tâches déjà allouées sur celui-ci. Cette hypothèse permet de nous concentrer sur l'ordonnançabilité pendant les transferts de migration. Nous supposons que nous connaissons statiquement si une tâche peut migrer, sur quel nœud et selon quelle échéance. Comme expliqué précédemment, cette hypothèse est valide dans le contexte de systèmes temps-réel critiques.

4.4 Techniques de migration temps-réel strictes

Dans cette section, nous présentons les politiques de migration qui exploitent le modèle de tâches et les hypothèses décrits précédemment. Comme nous connaissons la description statique du comportement temporel des tâches, les politiques proposées permettent de garantir les échéances de migration. Pour simplifier la description des algorithmes, nous considérons

l'échéance de migration à l'activation du prochain cycle de la tâche. Enfin, on note S les travaux système d'émission et R ceux de réception.

4.4.1 Copie Directe

Comme base de comparaison, nous avons adapté la politique classique de Copie Directe au contexte de systèmes temps-réel stricts. L'adaptation consiste à copier l'ensemble de l'espace mémoire de la tâche ($s(T_m)$) lors de l'une de ses périodes d'inactivité sur le nœud source, garantissant une terminaison des transferts avant l'activation du prochain cycle de la tâche. Les étapes suivantes doivent être complétées lorsque le dernier travail avant le prochain cycle a terminé son exécution :

1. Transférer une quantité de données équivalente à la taille mémoire de l'espace d'adressage de la tâche à migrer : $s(T_m)$;
2. Reprendre l'exécution de T_m sur le nœud de destination.

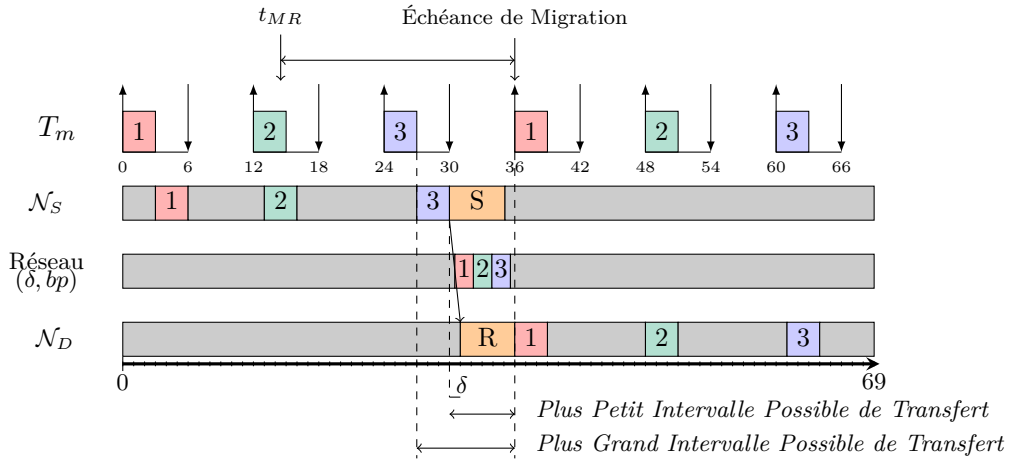


FIGURE 4.2 – Copie Directe.

Voici un exemple avec une tâche multiframe T_m telle que $T_m.E = \{3, 3, 3\}$ avec une échéance relative de $T_m.d = 6$ et une période de $T_m.p = 12$. Cette tâche va être migrée d'un nœud source $\mathcal{N}_S$ vers un nœud de destination $\mathcal{N}_D$. La demande de migration noté t_{MR} se produit pendant l'exécution du deuxième travail (au temps $t = 14$ sur la Fig. 4.2). L'échéance de migration est fixée au temps $t = 36$, de telle sorte que le premier travail de la tâche T_m sera exécuté à la reprise sur le nœud de destination. Le transfert ne commence que lorsque le troisième travail se termine (au temps $t = 30$). La *tâche système d'émission* T_σ se compose d'un travail S qui copie les données en relation avec T_m (i.e. celles des trois zones mémoires de ces travaux) sur le réseau, tandis que la *tâche système de réception* se compose d'un travail R qui les reçoit avec un délai δ. Lorsque le transfert est terminé, l'exécution de T_m est reprise sur son premier travail (au temps $t = 36$) sur le nœud de destination.

Nous présentons maintenant trois nouvelles techniques appelées *Prefetch Précopie*, *Postcopie* et Copie Mixte qui exploitent l'hypothèse selon laquelle l'espace modifié par chaque travail est connu statiquement hors-ligne. Nous introduisons j_m le travail courant, i.e. le travail s'exécutant lors de la demande de migration, et $s(j_m)$ son espace mémoire.

4.4.2 Prefetch Précopie

La *Prefetch Précopie* est adaptée de la technique classique de Précopie [TLC85] pour garantir que la quantité de données transférées est strictement égale à l'espace mémoire de la tâche. Aucun seuil n'est nécessaire pour détecter que certaines pages ont été modifiées. En effet, nous nous appuyons sur la connaissance statique du comportement temporel de la tâche à la place. La migration d'une tâche utilisant notre politique *Prefetch Précopie* peut donc être bornée dans le temps, permettant de fixer une échéance de migration à ce processus. De même, de par la propriété sur la mémoire des travaux (voir section 4.3), il est possible de savoir quelles pages ne seront pas modifiées par le travail courant. Les étapes suivantes doivent être réalisées :

1. Tant que le travail j_m est actif, commencer à copier les données correspondantes aux autres travaux déjà exécutés dans l'espace mémoire de $s(T_m) - s(j_m)$;
2. Si le travail courant est encore actif, attendre la fin de son exécution ;
3. Envoyer les pages restantes (au moins l'espace mémoire $s(j_m)$) au nœud de destination, puis reprendre l'exécution sur celui-ci.

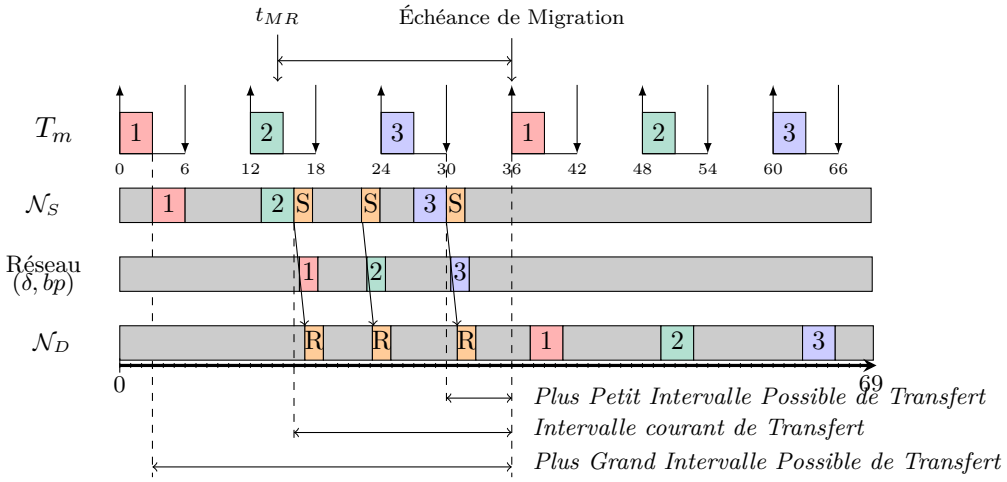


FIGURE 4.3 – Prefetch Précopie.

La Fig. 4.3 montre le plus petit et le grand intervalle de transfert possible pour cette technique ainsi que l'intervalle courant pour notre exemple décrit ci-dessus (voir section 4.4.1 page 62). Les transferts peuvent être divisés en deux parties. Premièrement, copier les données correspondant aux travaux déjà exécutés lorsque la demande de migration se produit, dans l'exemple le travail 1. Ensuite, transférer les travaux qui se terminent après la demande de migration, ici le deuxième et troisième travail. Dans cet exemple, j_m correspond au second travail de T_m. Nous pouvons constater que plusieurs travaux système d'émission sont utilisés pour transférer les données correspondant à la tâche en migration (T_m) : les travaux d'émission envoient les données de leurs travaux migrants respectifs. Tous les travaux doivent se terminer avant l'échéance de migration. De plus, soulignons sur l'exemple que les travaux système d'émission associés aux travaux 1 et 2 sont séparés : cela dépend de la politique d'allocation adoptée et de la charge sur le nœud $\mathcal{N}_S$.

4.4.3 Prefetch Postcopie

La *Prefetch Postcopie* consiste à copier la mémoire nécessaire pour l'exécution du premier travail suivant l'échéance de migration, puis à transférer toute la mémoire restante que la tâche

soit active ou non. De manière similaire à la *Prefetch Précopie*, cet algorithme dépend des propriétés sur la mémoire des travaux que nous avons décrit précédemment. Soit j'_m le premier travail suivant l'échéance de migration ($j'_m \neq j_m$) :

1. Transférer en premier les données correspondantes à $s(j'_m)$ avant la prochaine activation de la tâche que j_m soit actif ou pas ;
2. Lorsque l'espace mémoire de j'_m a été transféré et j'_m reprend son exécution sur le nœud de destination, transférer les pages restantes $s(T_m) - s(j'_m)$ en commençant par le travail suivant j'_m avant son activation.

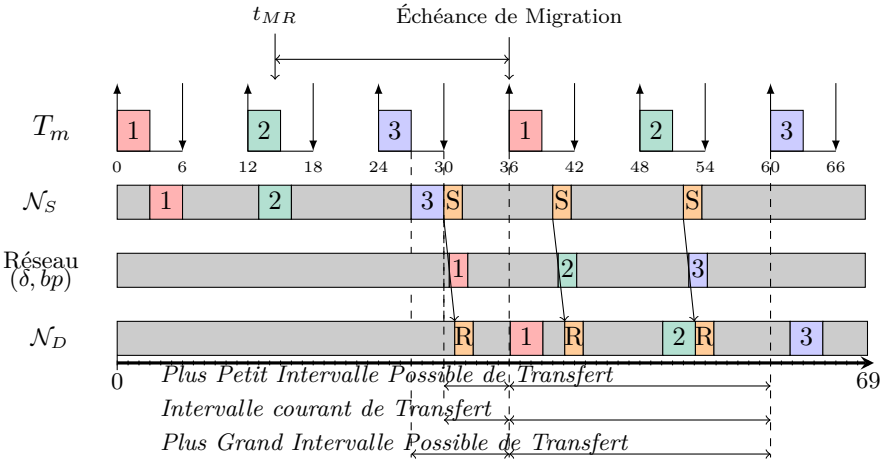


FIGURE 4.4 – Prefetch Postcopie.

Contrairement à la *Prefetch Précopie*, plusieurs travaux système peuvent être exécutés après l'échéance de migration, comme montré sur la Fig. 4.4. Cette figure montre le plus petit et plus grand intervalle de transfert possible ainsi que l'intervalle courant pour notre exemple. L'intervalle courant est ici le même que le plus petit intervalle car le dernier travail de la tâche migrante se termine au plus tard. Les transferts peuvent se diviser en deux parties. Premièrement, les transferts du premier travail qui sera exécuté sur le nœud de destination, ici j'_m est le premier travail de la tâche T_m, donc le premier à être envoyé. Ensuite, les transferts des travaux suivants, dans notre exemple les second et troisième travaux. Cependant, tous les travaux système d'émission doivent terminer avant l'activation de leur travail respectif sur le nœud de destination.

Cette dernière technique permet de reprendre l'exécution de la tâche migrée sur le nœud cible dès que l'espace mémoire nécessaire pour le travail suivant est disponible. Notons que *Prefetch Postcopie* copie ce qui est nécessaire pour les activations des prochains travaux à venir, tandis que *Prefetch Précopie* copie la mémoire des travaux qui sont terminés.

4.4.4 Copie Mixte

La Copie Mixte consiste à copier en premier la mémoire nécessaire pour le premier travail s'exécutant après l'échéance de migration, puis à transférer toute la mémoire restante que la tâche migrante ait ou non repris son exécution sur le nœud de destination. Cela peut être vu comme un mixte entre la *Prefetch Précopie* et la *Prefetch Postcopie*, donc l'algorithme dépend également des propriétés mémoires faites sur les travaux. Soit j'_m le *premier travail* s'exécutant après l'échéance de migration :

1. Transférer en premier les données correspondantes à $s(j'_m)$ avant la prochaine activation de la tâche que j_m soit actif ou non ;

2. Lorsque l'espace mémoire du prochain travail a été transféré, copier les pages restantes, $s(T_m) - s(j'_m)$, en commençant par le prochain travail suivant j'_m et avant son activation.

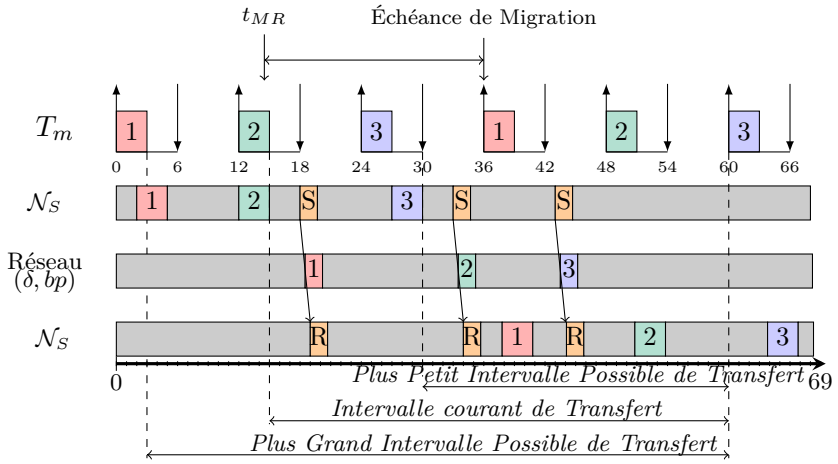


FIGURE 4.5 – Copie Mixte.

Comme montré sur la Fig. 4.5, cette technique peut mieux distribuer l'exécution des travaux système entre la demande de migration et la fin de la migration de la tâche. Cette figure montre le plus petit et plus grand intervalle de transfert possible ainsi que l'intervalle courant sur notre exemple. Les transferts peut être divisés en deux parties : les transferts se terminant à l'échéance de migration, ici ceux du premier travail de T_m et les transferts se terminant après l'échéance de migration, ici le second et troisième travaux. Notons que la deuxième partie des transferts peut commencer avant l'échéance de migration, comme montré sur la Fig. 4.5. Leurs exécutions courantes dépendront également des autres tâches présentes sur les nœuds.

Cette dernière technique reprend également l'exécution de la tâche migrée sur le nœud de destination dès que la mémoire nécessaire pour l'exécution du prochain travail est disponible. Contrairement à la *Prefetch Postcopie* qui envoie la majorité des données lorsque la tâche reprend son exécution, cette technique transfère les données nécessaires pour les prochaines activations des travaux sur le nœud de destination sans attendre.

4.5 Analyses hors-ligne

Dans le contexte des systèmes temps-réel stricts, il est nécessaire de fournir des garanties hors-ligne qu'au moins une des techniques peut être utilisée dans le pire scénario pour la migration d'une tâche. Nous parlerons alors de *faisabilité* de la migration. Cependant une technique de migration faisable ne sera pas forcément *ordonnançable* puisque dépendant de la charge apportée par les autres tâches du système. Il faut donc vérifier dans un premier temps la faisabilité de la migration en fonction des paramètres du réseau (notamment sa bande passante) et des durées d'inactivité de la tâche à migrer. Lorsque cela est fait, il restera à vérifier son ordonnançabilité avec le reste du système.

On note t_{MR} la date à laquelle la demande de migration arrive. Dans le cas général, une technique de migration est faisable *ssi* la tâche migrable T_m reprend son exécution sur le nœud de destination au plus tard à *la première activation* de travail après *l'échéance de migration*, i.e. $t \geq t_{MR} + D$ où D est le délai de migration défini par le concepteur. On suppose qu'un délai de migration minimal existe, nous le fixons comme étant égal à la période minimum d'inactivité de la tâche : $D_{min} = T_m.p - T_m.d$. Considérons maintenant des cas particuliers. La migration est définie comme faisable *au niveau de la tâche ssi* les transferts permettent l'exécution de la prochaine instance de la tâche (i.e. de son cycle) sur le nœud de destination. Dans ce cas, le délai de migration maximale est la durée du cycle : $D_{max} = N_j * T_m.p, N_j = |T_m.E|$. La migration est définie comme faisable *au niveau du travail ssi* les transferts permettent l'exécution du travail suivant sur le nœud de destination. Le délai de migration maximal est alors d'une période de la tâche : $D_{max} = T_m.p$.

Une demande de migration t_{MR} est immédiatement prise en compte par la tâche système entre le début de l'activation du *travail courant* et son échéance ($j_m.d$), voir Fig. 4.6. Lorsque t_{MR} se produit après l'échéance mais avant l'activation du travail suivant, la requête est repoussée à l'activation du prochain travail. Pour résumer, la requête de migration se produit sur des intervalles fixes définis par les activations et échéances des travaux de la tâche migrante :

$$k * T_m.p \leq t_{MR} \leq k * T_m.p + T_m.d, \text{ avec } k \in \mathbb{N}.$$

Lemme 1. *Le pire cas d'apparition de la requête de migration concernant sa faisabilité se produit à l'échéance du travail en cours, i.e.* $t_{MR} = k * T_m.p + T_m.d$ *où* $k \in \mathbb{N}$.

Démonstration. Considérons la faisabilité d'une migration au niveau du travail avec la technique de Copie Directe : cette technique de migration est faisable s'il existe suffisamment de temps pour envoyer $s(T_m)$, les données correspondant à la tâche migrante T_m, avant la prochaine activation de travail. Le délai de migration minimal correspond à une requête de migration à l'échéance du travail en cours : si cette technique est faisable pour ce délai, elle le sera pour tout délai plus long. Donc $t_{MR} = k * T_m.p + T_m.d$ où $k \in \mathbb{N}$ est bien le pire cas pour vérifier la faisabilité de la migration.
Le même raisonnement tient aussi pour la *Prefetch Précopie* et pour des migrations faisables au niveau de la tâche. Pour les autres techniques ce résultat peut être adapté en remplaçant le transfert de $s(T_m)$ par $s(j'_m)$, où j'_m est le prochain travail de T_m après l'échéance de migration. □

Lemme 2. *Le pire cas d'apparition de la requête de migration concernant l'ordonnançabilité se produit à l'échéance du travail en cours, i.e.* $t_{MR} = k * T_m.p + T_m.d$ *où* $k \in \mathbb{N}$.

Démonstration. Considérons maintenant l'ordonnançabilité pour une technique de migration faisable au niveau travail. Soit Γ_S et Γ_D les ensembles de tâches s'exécutant respectivement sur un nœud source $\mathcal{N}_S$ et sur un nœud de destination $\mathcal{N}_D$. Considérons le cas où la demande de migration t^{wc}_{MR} se produit à l'échéance du travail en cours pour une migration de T_m ($T_m \in \Gamma_S$) entre $\mathcal{N}_S$ et $\mathcal{N}_D$ à la prochaine activation de travail. Supposons qu'il existe un ordonnancement $\mathcal{S}_{WC}$ généré par un ordonnanceur optimal qui respecte toutes les échéances : celles de Γ_S et de T_σ la tâche système produite par t^{wc}_{MR}. La date de début de T_σ correspond à t_{MR}, la date de demande de migration. Le fait de considérer une demande de migration plus tôt, i.e. $t_{MR} \leq t^{wc}_{MR}$, ne fait que changer la date de début de la tâche système. En fait si la demande de migration arrive plus tôt, $t_{MR} \leq t^{wc}_{MR}$, il existe déjà un ordonnancement permettant de respecter toutes les échéances : celui généré pour t^{wc}_{MR} (noté $\mathcal{S}_{WC}$) car la tâche système correspondante a les mêmes caractéristiques d'exécution (à l'exception de sa date de début). Par la propriété d'optimalité

de l'ordonnanceur utilisé, nous savons que si un ordonnancement faisable existe, l'ordonnanceur optimal en trouvera également un. On a donc la garantie que s'il existe un ordonnancement pour t_{MR}^{wc}, il en existe pour toute demande de migration plus précoce : $\forall t_{MR} \leq t_{MR}^{wc}$. Donc, une demande de migration à l'échéance du travail peut être considéré comme le pire cas pour vérifier l'ordonnançabilité d'une tâche migrante sur un nœud source.
Notons que le même raisonnement tient pour la migration faisable au niveau de la tâche et pour vérifier l'ordonnançabilité au niveau du nœud de destination. □

4.5.1 Analyse de faisabilité

Nous présentons une analyse de faisabilité comme un problème de contraintes pour chaque technique de migration proposée et pour tout délai de migration $\forall D \geq D_{min}$. De plus, cet ensemble de contraintes permet de caractériser la tâche système T_σ en terme de temps d'exécution et d'échéances. Il s'agit de la première étape, une condition nécessaire supplémentaire, pour vérifier que l'ensemble des tâches reste ordonnançable pendant la migration. Nous rappelons que t_{MR} désigne la date de demande de migration et qu'elle se produit pendant une période d'inactivité de la tâche migrante (T_m).

Nous illustrons les conditions de faisabilité au niveau du travail pour chaque technique dans le meilleur et le pire cas de transfert, à l'aide de la Fig. 4.6. Sur celle-ci, la demande de migration se produit entre l'activation et l'échéance du travail j_m^1, l'échéance de migration est fixée à la période de la tâche migrante ce qui signifie que j_m^2 doit être repris sur le nœud de destination.

Le pire scénario pour que la Copie Directe soit faisable *au niveau du travail* est lorsque l'ensemble de l'espace mémoire de la tâche concernée T_m doit être transférée entre le pire cas d'apparition de la demande de migration et la période de la tâche (voir Fig. 4.6). Soit $k = \lfloor \frac{t_{MR}+D}{T_m.p} \rfloor$ le dernier intervalle avant l'échéance de migration, $\exists t_1, t_2 \in [t_{MR}, t_{MR}+D]$ tel que :

- $t_2 - t_1 = \frac{s(T_m)}{bp} \leq D_{min}$;
- $t_1 \geq k * T_m.p + T_m.d$ et $t_2 < (k+1) * T_m.p - \delta$;
- $T_\sigma.e = t_2 - t_1$ et $T_\sigma.d = (k+1) * T_m.p - \delta$.

La dernière contrainte exprime les propriétés de la tâche système i.e. son temps d'exécution et son échéance relative.

Le pire scénario pour que la *Prefetch Précopie* soit faisable *au niveau du travail* est identique au scénario de la Copie Directe (voir Fig. 4.6). En effet, les transferts s'effectuent alors dans le même intervalle correspondant à la durée minimale d'inactivité de la tâche migrante D_{min}.
Dans le cas général, $\forall D \geq D_{min}$: une condition suffisante est de trouver suffisamment de temps, alloué à la tâche système $T_\sigma = J_{\sigma_1} \cup \{j_{\sigma_2}\}$, tel que la majorité de l'espace mémoire puisse être transféré avant la dernière période d'inactivité de T_m avant l'échéance de migration, effectué par l'ensemble des travaux J_{σ_1}. Et tel que l'espace mémoire restant soit transféré pendant D_{min} par le travail j_{σ_2}. Soit $s(j_m)$ la taille du dernier travail avant l'échéance de migration, $\forall j_m \in T_m, k = \lfloor \frac{t_{MR}+D}{T_m.p} \rfloor$, $\exists t_1, t_2, t_3 \in [t_{MR}, t_{MR}+D]$ tel que :

- $t_2 - t_1 \geq \frac{s(T_m)-s(j_m)}{bp}$, $t_3 - t_2 = \frac{s(j_m)}{bp} \leq D_{min}$ et $t_3 - t_1 \leq D$;
- $t_2 \geq k * T_m.p + T_m.d$ et $t_3 \leq (k+1) * T_m.p - \delta$;

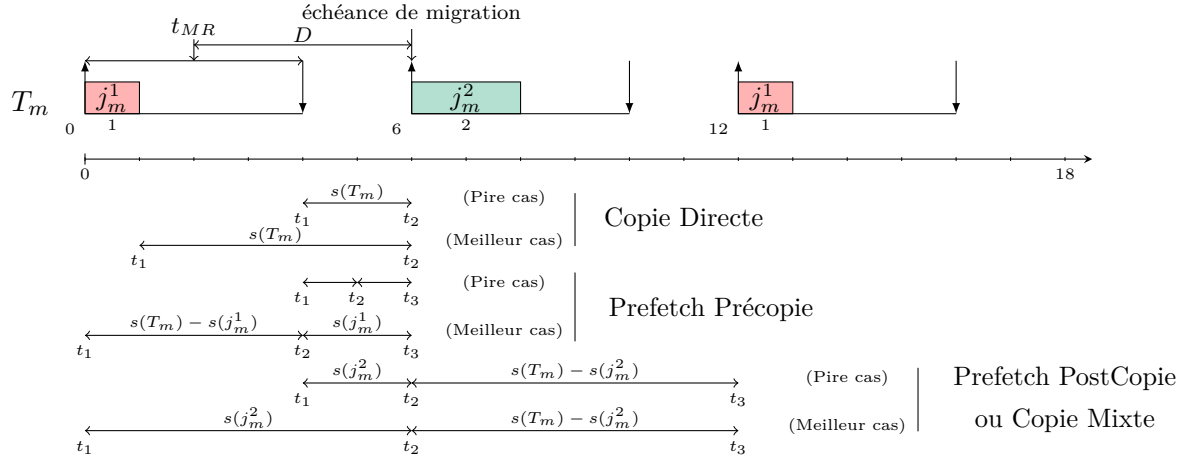


FIGURE 4.6 – Intervalles de migration possibles *au niveau du travail* dans les meilleurs et pires cas avec une tâche mutiframe de 2 travaux $T_m = (\{1,2\},4,6)$. Les instants t_1, t_2, t_3 correspondent aux bornes de transfert pour chaque technique. A l'exception de la Copie directe, l'instant t_2 est fixé à titre indicatif et ne représente pas forcément la réalité des transferts.

- $\sum_{j_{\sigma_1} \in J_{\sigma_1}} j_{\sigma_1}.e = \frac{s(T_m)-s(j_m)}{bp}$ et $\forall j_{\sigma_1} \in J_{\sigma_1}, j_{\sigma_1}.d = (k+1) * T_m.p - \delta$
  et $j_{\sigma_2}.e = \frac{s(j_m)}{bp}, j_{\sigma_2}.d = (k+1) * T_m.p - \delta$.

Dans le cas général décrit ci-dessus, plusieurs travaux s'exécutent et ont des caractéristiques bien précises mais partagent la même échéance. Dans le pire des cas, la tâche système est caractérisée de la même manière que la Copie Directe, voir le problème de contraintes de la Copie Directe.

Le pire scénario pour que la *Prefetch Postcopie* soit faisable *au niveau du travail* est lorsque l'espace mémoire du prochain travail de T_m doit être transféré entre le pire cas d'apparition de la demande de migration et la période de la tâche (voir Fig. 4.6). Une condition suffisante est de trouver suffisamment de temps, alloué à une tâche système $T_\sigma = \{j_{\sigma_1}\} \cup \Gamma_{\sigma_2}$, tel que : 1) le travail système j_{σ_1} puisse transférer la mémoire du travail suivant avant l'activation de celui-ci, 2) l'ensemble de travaux système Γ_{σ_2} puisse transférer l'espace mémoire restant avant la fin du cycle de la tâche, 3) les parties restantes sont transférées avant l'activation de leur travail respectif. Soit $s(j'_m)$ la taille du premier travail après l'échéance de migration et $j^i_{\sigma_2}$ le $i^{\text{ème}}$ travail de J_{σ_2}, $\forall j'_m \in T_m, k = \lfloor \frac{t_{MR}+D}{T_m.p} \rfloor$, $\exists t_1, t_2, t_3 \in [t_{MR}, t_{MR}+D]$ tel que :

- $t_2 - t_1 = \frac{s(j'_m)}{bp} \leq D_{min}$, $t_3 - t_2 \geq \frac{s(T_m)-s(j'_m)}{bp}$, $t_3 - t_1 \leq N_j * T_m.p$;
- $t_1 \geq k * T_m.p + T_m.d$ et $t_2 \leq (k+1) * T_m.p - \delta$;
- $j_{\sigma_1}.e = \frac{s(j'_m)}{bp}$, $j_{\sigma_1}.d = (k+1) * T_m.p - \delta$
  et $\sum_{j_{\sigma_2} \in J_{\sigma_2}}, j_{\sigma_2}.e = \frac{s(T_m)-s(j'_m)}{bp}, \forall i \in [1, N_j - 1], j^i_{\sigma_2}.d = (k+1+i) * T_m.p - \delta$.

Le pire cas pour la Copie Mixte est le même que pour la *Prefetch Postcopie*. L'analyse de faisabilité est donc similaire à celui de la *Prefetch Postcopie* : la seule différence se situe sur le fait que les dates d'activation des travaux système peuvent êtres fixée au plus tôt à la fin de l'exécution du travail qu'ils doivent transférer, leur temps d'exécution et leur échéance relative restant inchangés.

Discussion

Nous avons montré que les résultats de faisabilité de la Copie Directe et de la *Précopie* sont identiques dans le pire des cas au niveau du travail. Les politiques de *Postcopie* et de Copie Mixte permettent plus de flexibilité que ce soit dans le meilleur et dans le pire des cas, car la majorité de l'espace mémoire peut être transférée après que le tâche migrée ait repris son exécution sur le nœud de destination, indépendamment de la date de demande de migration t_{MR}. Ces politiques sont celles qui nécessitent la durée d'inactivité la plus courte pour être faisable (voir Fig. 4.6).

On considère maintenant la migration au niveau de la tâche. Les conditions de faisabilité de la Copie Directe sont identiques à celles au niveau du travail car le transfert doit toujours s'effectuer sur la dernière période d'inactivité avant l'échéance de migration, voir Fig. 4.2. Les conditions de faisabilité de la politique de *Précopie* sont également identiques à celles au niveau du travail mais seulement dans le pire des cas : dans le cas général, l'intervalle de transfert est plus grand permettant l'utilisation de plusieurs travaux système, voir Fig. 4.3. Concernant les politiques de *Postcopie* et de Copie Mixte, les conditions de faisabilité diffèrent de celles au niveau travail par le temps supplémentaire disponible pour copier les pages avant la prochaine instance de la tâche que ce soit dans le meilleur ou le pire des cas, voir Fig. 4.4 et Fig. 4.5. Remarquons également que les conditions de faisabilité dans le meilleur cas pour *Précopie* au niveau de la tâche sont équivalentes à celles de *Postcopie*. Enfin, le plus grand intervalle de transfert possible est obtenu dans le meilleur des cas pour la faisabilité au niveau de la tâche de la Copie Mixte.

Dans le meilleur des cas, les conditions de faisabilité dépendent de l'avancement de l'exécution de la tâche à migrer ainsi que de la politique employée.

4.5.2 Analyse d'ordonnançabilité

Dans cette section, nous montrons comment intégrer la charge générée dans le pire des cas par les tâches systèmes en charge des migrations dans des tests d'ordonnançabilité CPU et réseau.

Baruah et al. [BCGM99] ont présenté un test d'ordonnançabilité pour les tâches multiframe générales qui se base sur le concept de demande processeur cumulative (*demand bound function*), que nous noterons $dbf(T,t)$. Cette fonction correspond à l'exécution cumulative de chaque travail de chaque tâche qui ont leurs dates de début et échéances absolues dans l'intervalle de durée $[0;t]$. Lorsqu'on utilise un modèle de tâches multiframe, la fonction de demande processeur augmente de la valeur du temps d'exécution du travail courant pour chaque intervalle atteignant une valeur multiple de $T.p$. [BCGM99] propose un algorithme pour calculer cette fonction pour chaque tâche. Un ensemble de tâches Γ est faisable si et seulement si $\sum_{T\ in\Gamma} dbf(T,t) \leq t$ pour une certaine valeur positive de t. Les résultats s'appliquent pour le modèle de tâche multiframe que nous considérons dans ce travail. Nos mécanismes de migrations ajoutent une tâche système avec offset, donc la fonction cumulative doit être nécessairement vérifiée pour un ensemble de tâches asynchrones.

Les caractéristiques précises de la tâche système (temps d'exécution, échéance et date de début) dépendent de la technique de migration retenue. Les dates de début dépendent de la fréquence de migration au pire qui est fixée pour une tâche migrable. Ce paramètre est sous le contrôle du concepteur du système distribué temps-réel strict. La section 4.5.1 nous a permis de spécifier les temps d'exécution et échéance de la tâche système pour une tâche migrable.

Pour les tests d'ordonnançabilité, le concepteur du système doit envisager les résultats pour

le pire cas de demande de migration et pour la fréquence de migrations de tâche envisagée (f). Toutes les techniques de migrations présentées ici ne peuvent prétendre à la même fréquence. Pour illustrer ce propos, nous présentons la fréquence *maximale* possible de ces techniques (f_{max}). Pour les politiques de Copie Directe (Fig. 4.7-a) et de *Prefetch Précopie* (Fig. 4.7-b), f_{max} est égale à $T_m.p$ (la période de la tâche migrante). Elle peut être implémentée en utilisant une tâche système périodique, i.e. une tâche multiframe avec un seul travail, dont la date de début se situe à l'échéance du travail de la tâche migrante. Pour la politique de *Prefetch Postcopie* (Fig. 4.7-c), nous considérons que f_{max} est égal au cycle de la tâche : la tâche système est équivalente à une tâche multiframe qui délivre un premier travail avec un offset égal à l'échéance du dernier travail de la tâche migrante avant l'échéance de migration et délivre les autres travaux système à l'échéance de migration (i.e. avec un temps de séparation nul). Pour la politique de Copie Mixte (Fig. 4.7-d), une tâche multiframe peut être définie de manière similaire à la *Prefetch Postcopie* à l'exception que nous considérons f_{max} égale à deux cycles de la tâche migrante pour éviter la concurrence d'exécution entre des travaux systèmes de deux demandes de migration différentes. Une tâche multiframe peut donc être ajoutée au test d'ordonnançabilité CPU basé sur la demande processeur cumulative comme tout autre tâche, et pour chacune des techniques proposées. Notons aussi que dans une implémentation réelle, une migration peut commencer dès que le travail d'une tâche migrante se termine.

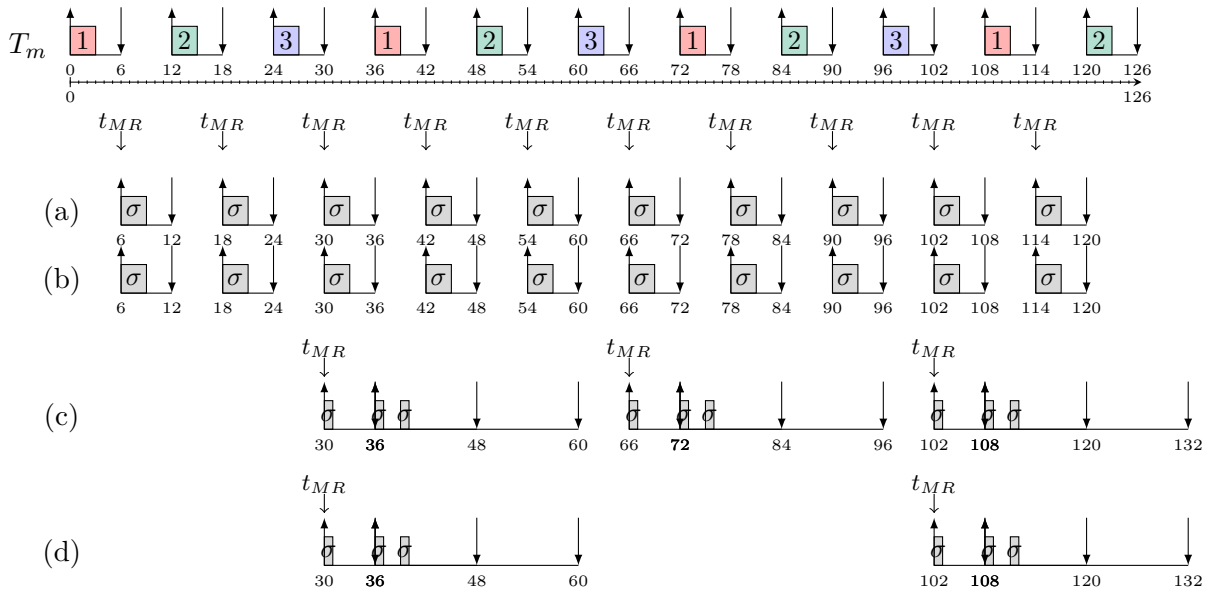


FIGURE 4.7 – Caractérisation du pire cas d'activation de la tâche système correspondant à sa fréquence maximale selon la politique envisagée : a) Copie Directe, b) Prefetch Precopie, c) Prefetch Postcopie et d) Copie Mixte.

Pour un test d'ordonnançabilité CPU et réseau, l'un des tests bien connu est l'approche par offset dynamique décrite dans [PGH98] (voir également le chapitre 2 pour une description de l'approche). Même si cette méthode se concentre initialement sur l'ordonnancement à priorité fixe, Rahni et al. [RGR08] l'ont étendu à l'ordonnancement à priorité dynamique (type EDF). Dans ces travaux, ils utilisent un modèle de tâches temps-réel appelé *transaction* : ce modèle convient bien pour les communications distantes entre les nœuds en définissant gigues et offsets. Nous pouvons utiliser ce type de test d'ordonnançabilité dans notre contexte en remarquant qu'une tâche multiframe est un cas particulier d'une transaction : une transaction avec une gigue nulle. L'ajout de ces tâches additionnelles pour les migrations de tâche dans ce type de

test d'ordonnançabilité peut être réalisé en connaissant la taille des messages. Dans notre cas, elle correspond aux zones mémoires de la tâche migrante.

4.6 Politique de choix en-ligne

Après avoir vérifier que l'une des techniques de migration est faisable et ordonnançable, il reste encore quelques paramètres importants qui peuvent être déterminés en ligne.

4.6.1 Caractéristiques des travaux systèmes

Une fonction possible de la politique en-ligne est de distribuer la mémoire à transférer (correspondant à une charge CPU et réseau) de manière équitable entre chaque travail système. En effet, nous avons fait l'hypothèse jusque là que les espaces mémoires des travaux étaient identiques i.e. $\sum_{j_m \in T_m} s(j_m) = |T.E| * s(j_m) = s(T_m)$. Si les espaces mémoires des travaux de la tâche migrante ne sont pas de même taille, il serait intéressant de diviser les données transférées proportionnellement au nombre de travaux système disponibles. Ceci permettrait dans le meilleur des cas un bon équilibre de la charge réseau et CPU induite par la migration de la tâche.

Une méthode simplement applicable est envisageable telle que le premier travail transfère tout l'espace déjà exécuté, les suivants se contentent de transférer uniquement le dernier travail exécuté. Pour tout t_{MR} l'instant de la demande de migration, la procédure suivante doit déterminer :

1. le nombre de travaux nécessaires N_σ ;
2. pour chacun, leur date de début au plus tôt $j_\sigma.s$, leur temps d'exécution $j_\sigma.e$ et leur échéance $j_\sigma.d$.

Détermination du nombre de travaux systèmes

Pour la *Prefetch Postcopie* et la Copie Mixte, le nombre de travaux ne varie pas, il est de $N_\sigma = N_j = |T_m.E|$. Pour la Copie Directe, il y a un seul travail système ($N_\sigma = 1$). Contrairement aux autres techniques de migrations, la *Prefetch Précopie* utilise un nombre variable de travaux systèmes pour transférer les données sur le réseau. Il est nécessaire de configurer ce nombre en-ligne, dès que la demande de migration est connue car ce nombre dépend de la date de requête de migration.

Concernant la *Prefetch Précopie*, deux cas se présentent : soit k la période de la tâche migration où apparaît la demande de migration $k = \lfloor \frac{t_{MR}}{T_m.p} \rfloor mod\ N_j$,
Cas 1 ($D < T_m.p$) : si le délai de migration est inférieur à la période de la tâche migrante et j_m^k termine son exécution alors on considère qu'un seul travail système est nécessaire $|T_\sigma.E| = 1$ tel que $j_\sigma.e = \frac{s(T_m)}{bp}$ et $j_\sigma.d = T_m.p - t_{MR} - \delta$

Cas 2 : $T_\sigma = \{\{j_\sigma^1, j_\sigma^2, ..., j_\sigma^l\}, j_\sigma^{N_\sigma}\}$, $\{j_\sigma^1, j_\sigma^2, ..., j_\sigma^l\}$ est l'ensemble des travaux précopiant les données, $j_\sigma^{N_\sigma}$ est le travail pour le dernier transfert avant reprise de la tâche migrée sur le nœud de destination. Une façon simple de caractériser le nombre de travaux est selon l'endroit dans le cycle de la tâche migrante où la demande de migration se produit : $|T_\sigma.E| = N_\sigma = \lceil \frac{D}{T_m.p} \rceil, N_\sigma \in [2, N_j]$. Un nombre $(N_\sigma - 1)$ travaux servent à précopier et un dernier travail est utilisé pour le transfert juste avant l'échéance de migration. Chaque travail système j_σ commence son exécution au mieux à la fin du travail lui étant attribué.

Répartition des données transférées

On dispose de l'échéance de migration, par conséquent du délai de migration D, des caractéristiques de la tâche à migrer T_m $(T_m.p, T_m.d, T_m.E)$ et de la taille mémoire des travaux $\forall j \in T_m, s(j)$. Nous présentons deux algorithmes qui pourraient être exécutés pour la *Prefetch Précopie* à l'instant où la demande de migration est connue.

La façon la plus simple est d'affecter un premier travail système pour le transfert de tous les travaux déjà exécutés et aux autres travaux système un travail en cours de la tâche migrante. Soit j_σ^i le $i^{\text{ème}}$ travaux système de T_σ, $(j_\sigma^i.s, j_\sigma^i.e, j_\sigma^i.d)$ respectivement sa date de début et son temps d'exécution, et soit $j.f$ la date de fin d'exécution d'un travail j :

Algorithme Entrées : $\{D, t_{MR}, T_m\}$, Sortie : T_σ

$k \leftarrow \lfloor \frac{t_{MR}}{T_m.p} \rfloor \mod N_j$
$N_\sigma \leftarrow \lceil \frac{D}{T_m.p} \rceil$
$j_\sigma^1.s \leftarrow \min(t_{MR}, j_m^k.f)$
$j_\sigma^1.e \leftarrow \sum_{i \in [1,k]} \frac{s(j_m^i)}{bp}$
$j_\sigma^1.d \leftarrow t_{MR} + D - \delta$
pour i de 2 à N_σ **faire**
    $j_\sigma^i.s \leftarrow j_m^{k+i-1}.f$
    $j_\sigma^i.e \leftarrow \frac{s(j_m^{k+i-1})}{bp}$
    $j_\sigma^i.d \leftarrow t_{MR} + D - \delta$
fin pour

L'algorithme ci-dessus ne permet donc pas de répartir équitablement les transferts, le premier travail système transférant potentiellement plus de données : il démarre dès la fin du travail courant et transfère tous les travaux déjà exécutés. Puis les autres travaux systèmes transfèrent au fil de l'exécution des travaux de la tâche migrante.

Pour répartir plus équitablement la charge sur les différents travaux système, on propose d'affecter en moyenne un temps d'exécution égal à $\frac{s(T_m)}{N_\sigma * bp}$ sachant que chacun des j_σ ne peut transférer que des données correspondant à des travaux déjà exécutés. Soit k la période surlaquelle apparaît la demande de migration, N_σ le nombre de travaux système nécessaires, $k + N_\sigma - 1$ correspond à l'indice du dernier travail de la tâche migrante avant l'échéance de migration. Donc on propose d'affecter au plus un temps d'exécution égal à $\frac{s(T_m) - s(j_m^{k+N_\sigma-1})}{(N_\sigma - 1) * bp}$ aux $N_\sigma - 1$ premiers travaux systèmes.

Algorithme Entrées : $\{D, t_{MR}, T_m\}$ Sortie : T_σ

$\mathcal{S} \leftarrow \{s(j_m^1), s(j_m^2), ..., s(j_m^{N_j})\}$
$k \leftarrow \lfloor \frac{t_{MR}}{T_m.p} \rfloor \mod N_j$
$N_\sigma \leftarrow \lceil \frac{D}{T_m.p} \rceil$
$\{j_\sigma^1.e, ..., j_\sigma^{N_\sigma}.e\} \leftarrow \{0, ..., 0\}$
$l \leftarrow 0$
$q \leftarrow \frac{s(T_m) - s(j_m^{k+N_\sigma-1})}{(N_\sigma - 1) * bp}$
▷ *Caractéristiques des $N_\sigma - 1$ travaux systèmes (si $N_\sigma > 1$)*
pour i de 1 à $N_\sigma - 1$ **faire**
    tant que $j_\sigma^i.e < q$ et $l \leq k + i - 1$ et $N_\sigma > 1$ **faire**
        $j_\sigma^i.e$ += $\min(\frac{s(j_m^l)}{bp}, q - j_\sigma^i.e)$
        $S[l]$ -= $\min(\frac{s(j_m^l)}{bp}, q - j_\sigma^i.e)$
        $j_\sigma^i.s \leftarrow j_m^l.f$
        si S[l] = 0 **alors faire**
            l++

fin si
fin tant que
$j_\sigma^i.d \leftarrow t_{MR} + D - \delta$
fin pour
▷ *Caractéristiques du dernier travail système*
$j_\sigma^{N_\sigma}.s \leftarrow j^{k+N_\sigma-1}.f$
si $N_\sigma > 1$ **alors faire**
$j_\sigma^{N_\sigma}.e \leftarrow s(j_m^{k+N_\sigma-1})$
fin si
sinon
$j_\sigma^{N_\sigma}.e \leftarrow \sum_{i\in[1,N_j]} \frac{s(j_m^i)}{bp}$
fin si
$j_\sigma^{N_\sigma}.d \leftarrow t_{MR} + D - \delta$

Dans cet algorithme, $\mathcal{S}$ est un tableau contenant la taille mémoire de chacun des travaux de la tâche migrante. On définit une quantité de données q que chaque travail système ne doit pas dépasser, elle correspond à la moyenne des tailles des $N_j - 1$ premiers travaux. Pour chaque travail, on associe un temps d'exécution au plus égal à cette quantité. Pour chaque travail à transférer et associé à un travail système, la case correspondante dans le tableau $\mathcal{S}$ devient nulle et l, correspondant à l'avancement des travaux transférés, augmente d'une unité. Le dernier travail système $j_\sigma^{N_\sigma}$ transfère toujours les données correspondant à l'espace mémoire du dernier travail $j_m^{N_j}$ avant l'échéance de migration sauf s'il n'y a qu'un seul travail système : dans ce cas, on transfère l'intégralité de l'espace mémoire de la tâche migrante.

Une autre fonction de la politique en-ligne peut être de choisir une technique parmi plusieurs démontrées comme étant faisables. De même, si une technique n'est pas faisable, elle pourrait être choisie en-ligne si le scénario permet de garantir les échéances : par exemple si la Copie Directe n'est pas faisable pour la durée minimale d'inactivité D_{min}, il est possible de rencontrer un cas où la durée d'inactivité est supérieure et suffisante pour l'emploi de cette technique.

4.7 Évaluation de performance des techniques de migrations

Afin d'estimer l'intérêt pratique de nos techniques de migration, nous avons effectué un ensemble d'exécutions grâce à un simulateur d'ordonnancement développé en interne. Nos techniques de migrations disposent déjà, selon le jeu de tâches considéré, de garanties hors-ligne pour vérifier leur ordonnançabilité. Nous pouvons donc produire des ordonnancements corrects, respectant toutes les échéances, avec ou sans migrations de tâche. Nous devons trouver un moyen d'évaluer leur performance lors de l'exécution. Nous avons choisi de comparer les ordonnancements produits en termes de surcoûts de préemptions de travaux induits par les politiques de migration tâches proposées. Comme nous considérons un ordonnancement préemptif, le nombre de changements de contextes doit être pris en compte dans le WCET. Notre simulateur évalue ce nombre de préemptions de travaux dû aux techniques de migration en fonction de la fréquence de migration de la tâche migrable, du nombre de travaux la constituant et de la taille de son espace mémoire.

4.7.1 Configuration de la simulation

Nous utilisons EDF comme ordonnanceur par défaut. Comme il est optimal en monoprocesseur, nous nous concentrerons sur les effets de la migration de tâches sur un seul nœud, ici le nœud source. Par conséquent, nous considérons seulement des tâches non-communicantes. Nous fixons la bande passante réseau *bp* à 100 Mbps, une valeur commune pour un réseau temps-réel [ARI02] et nous considérons que le délai de transmission est négligeable ($\delta = 0$).

Pour chaque pas des paramètres envisagés, nous nous proposons d'exploiter le résultat de 100 jeux de tâches générés aléatoirement. Sur chaque jeu de tâche, l'une d'elle est choisie pour être la tâche migrante. D'une manière générale, un jeu de tâches Γ est constitué de huit tâches multiframe : cette valeur est relativement basse mais courante dans un contexte temps-réel critique [DAL+04, JDLP10]. Nous considérons :

- une unique tâche migrable :
  $T_m = \{T_m.E, T_m.d, T_m.p, s(T_m)\}$ avec des paramètres constants tels que $\forall j_m \in T_m, j_m.e < T_m.d$. Nous choisissons $T_m.d < T_m.p$ et la taille mémoire est fixée à une valeur de 1 Mb.

- sept autres tâches : tous les paramètres sont des variables réelles, générées aléatoirement avec une distribution uniforme tel que $\forall T \in \Gamma - \{T_m\}, \forall j \in T, j.e < T.d$ and $T.d < T.p$.

Pour toutes les tâches, la période $T.p$ est déterminée aléatoirement suivant une loi uniforme entre $p_{min} = 100$ et $p_{max} = 1000$. L'échéance relative $T.d$ est comprise entre p_{min} et $0.8{*}T.p$. Les temps d'exécution d'exécution sont tirés entre 1 et $0.6{*}T.d$. Le taux global d'utilisation retenu doit être strictement supérieur à 50% : $U_{tot} > 0.5$. La raison principale est que si le jeu de tâches avait un taux d'utilisation trop faible, les migrations de tâches pourraient ne pas influer sur l'exécution, par exemple si la tâche système s'exécutait lorsque toutes les autres tâches sont inactives. Enfin, la taille de l'espace d'adressage de la tâche migrante $s(T_m)$ est telle que $\frac{s(T_m)}{bp} \leq T_m.p - T_m.d$.

Nous générons des jeux de tâches *faisables* pour toutes nos techniques de migrations et pour la fréquence maximale de la Copie Directe : ces jeux permettent donc l'utilisation des techniques les plus exigeantes en termes de temps d'inactivité nécessaire (Copie Directe et Précopie) pour des demandes de migration à chaque période la tâche migrante ($T.p$). Pour toutes ces techniques, nous considérons le plus petit intervalle de transfert comme montré en section 4.4. De plus, nous définissons un autre paramètre : le nombre de travaux constituant chacune des tâches multiframe, i.e. $N_j \in \{2, 4, 8\}$. Nous rappelons qu'une technique de migration est implémentée à l'aide d'une tâche système additionnelle comme défini en section 4.5.2.

Nous définissons la longueur de l'ordonnancement par L, suffisamment grande pour pouvoir effectuer toutes les migrations de tâche. Nous fixons le nombre de migrations, noté N_{mig}, entre 0 et $L/T_m.p$ (ce qui correspond à f_{max} i.e. une par période de la tâche). Dans un premier temps, nous évaluons les préemptions produites par EDF sans migration. Cela nous permet d'obtenir un ordonnancement de référence, p_{ref}, sur le nombre de préemptions de travaux uniquement dues à la politique d'ordonnancement. Nous notons p_{mig} le nombre de préemptions de travaux produites avec EDF en présence de migration de tâches. Pour comparer l'efficacité des différentes techniques entre elles et par rapport à une exécution sans migration, nous définissons un ratio de préemptions p_{mig} / p_{ref}.

4.7.2 Résultats de la simulation

La première partie de notre étude expérimentale se concentre sur la fréquence de migration. Lorsque la fréquence f est fixée à une toutes les cycles ou toutes les périodes, les requêtes de migration se produisent à chaque cycle ou période de la tâche. Dans tous les autres cas, les requêtes de migration sont distribuées uniformément sur la longueur de l'intervalle d'exécution L. La Figure 4.8 montre le ratio moyen de préemptions en fonction du nombre de migrations, pour des tâches multiframe composée de deux travaux ($N_j = 2$). Comme attendu, ce ratio augmente avec la fréquence de migration. Nous pouvons voir que pour un nombre de migrations en dessous de 30, l'augmentation du ratio n'excède pas 1% quelque soit la technique utilisée. Pour $N_{mig} = 100$, les techniques de migration commencent à se différencier de manière notable. Le plus

grand surcoût est proche de 14%, et n'apparaît seulement avec la Copie Directe, la seule utilisée pour la fréquence maximale. Utiliser les autres techniques lorsque les demandes de migration se produisent à chaque période de la tâche n'a en effet pas de sens dans notre contexte : toutes les techniques opéreraient de manière similaire à la Copie Directe. La technique apparaissant comme la plus efficace est la Copie Directe, suivie de la *Prefetch Précopie*, la Copie Mixte et la *Prefetch Postcopie*. Notons qu'une fréquence au cycle de la tâche migrante (respectivement à la période de la tâche) correspond dans notre exemple à $N_{mig} = 500$ (respectivement à $N_{mig} = 1000$). Concernant la Copie Mixte, la courbe s'arrête avec une valeur de N_{mig} légèrement inférieur à 250. Il est en effet impossible de trouver une valeur de N_{mig} supérieure à 250 telle que les requêtes de migration se produisent tous les deux cycles de la tâche (la fréquence maximale définie pour cette technique). C'est un prérequis pour la politique de Copie Mixte, car le fait d'autoriser des requêtes de migration plus proches pourrait amener à entrelacer les travaux systèmes de deux requêtes différentes. Ce comportement pourrait impliquer le dépassement de certaines échéances.

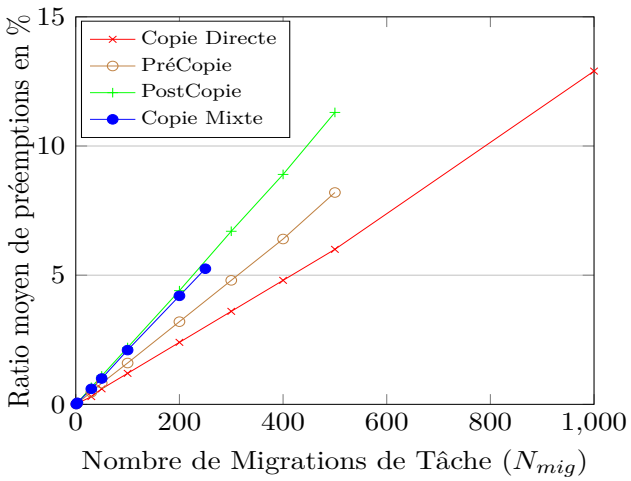


FIGURE 4.8 – Ratio moyen de préemptions de travaux fonction du nombre de migrations de tâches ($N_j = 2$).

Dans une seconde partie, nous nous concentrons sur un autre paramètre intéressant : le nombre de travaux par tâche (N_j). Nous avons fixé le nombre de migrations à 100 pour être sûr de voir apparaître des différences significatives. A l'exception de la Copie Directe, toutes les techniques utilisent la distinction de l'espace mémoire des travaux pour distribuer les transferts de migration. Comme attendu, faire varier le nombre de travaux par tâche n'a aucune incidence sur les résultats de la Copie Directe, comme le montre la Figure 4.9. Pour la *Prefetch Précopie*, nous observons une très légère augmentation, ce qui s'explique principalement parce que cette technique n'utilise pas un nombre constant de travaux (pas toujours le nombre maximum par rapport aux autres techniques) dépendant de la date de la requête de migration. La *Prefetch Postcopie* est celle qui produit le plus de préemptions supplémentaires. La Copie Mixte est très proche pour $N_j = 2$, mais sa courbe se situe entre celle de *Prefetch Postcopie* et celle de la *Prefetch Précopie* pour des valeurs supérieures de N_j ($\{4, 8\}$). Ceci peut s'expliquer par la plus grande latitude dans l'exécution des travaux système de la Copie Mixte par rapport à ceux de la *Prefetch Précopie* (voir les longueurs d'intervalle possible de transfert sur les Fig. 4.4 et 4.5).

Enfin, un dernier paramètre important de la migration est la taille de l'espace d'adressage ($s(T_m)$). Jusqu'ici nous avons utilisé une taille unique de 1 Mb. Pour compléter notre expérimentation, nous nous sommes basés sur les données fournies par Guthaus et al. [GRE+01] comparant

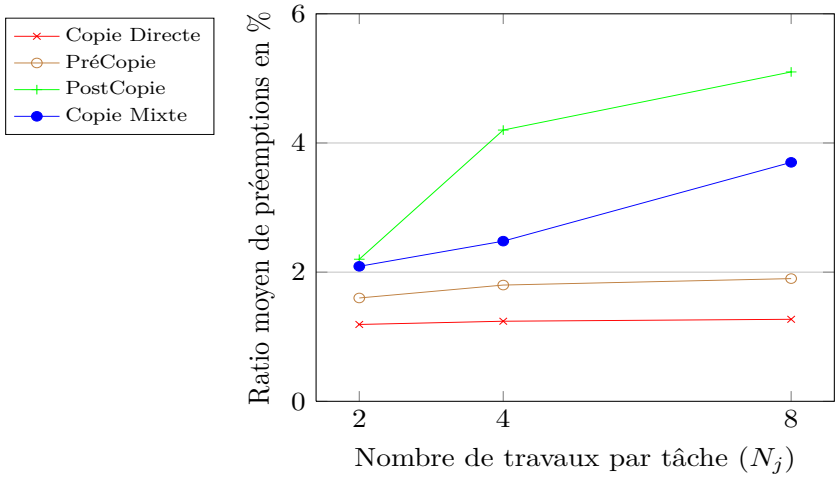


FIGURE 4.9 – Ratio moyen de préemptions fonction du nombre de travaux par tâche ($N_{mig} = 100$).

des benchmarks représentatifs des applications embarqués commerciales. L'une des constatations que l'on peut effectuer est que les applications ont une empreinte mémoire globalement comprise entre 50 ko et 250 ko. La taille prise par défaut dans les expérimentations précédentes rentre bien dans cette fourchette. Nous nous proposons maintenant de voir l'influence de la taille mémoire de la tâche migrante dans cette fourchette de valeurs.

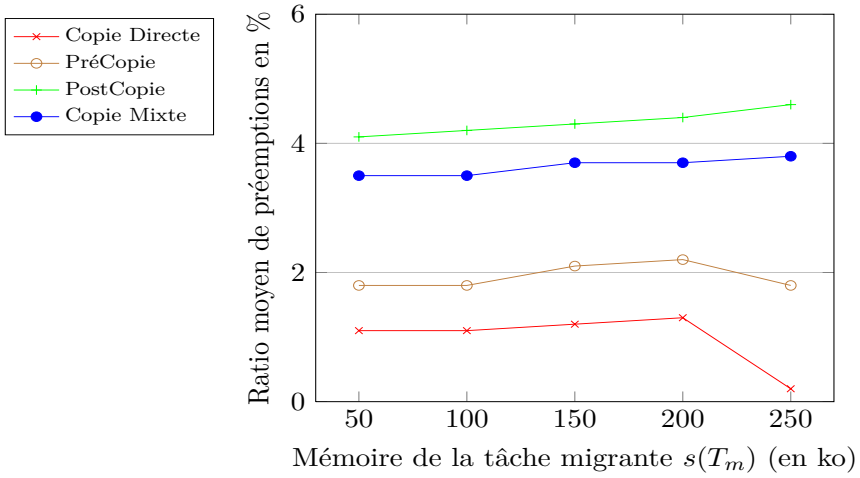


FIGURE 4.10 – Ratio moyen de préemptions fonction de la taille mémoire de la tâche migrante ($N_{mig} = 100$, $N_j = 4$).

La Fig. 4.10 présente l'évolution des surcoûts de préemptions suivant la taille mémoire de la tâche migrante pour les quatre techniques proposées. Nous avons choisi de fixer N_{mig} à 100 et N_j à 4 afin de distinguer nettement les différences entre les techniques. D'une manière générale, l'allure confirme les résultats précédemment observés : la *Prefetch Postcopie* est la plus coûteuse suivie de la Copie Mixte, la *Prefetch Précopie* et le Copie Directe. Pour des valeurs de taille mémoire entre 50 et 200 ko la courbe est en très légère augmentation : plus la taille mémoire

augmente et plus le temps d'exécution de la tâche système associée est important. Plus ce temps est important et plus il existe de chances pour que la tâche système influe sur l'exécution globale, ce qui explique qu'il y ait en moyenne légèrement plus de préemptions.

Pour $s(T_m) = 250$ ko, nous rencontrons un cas très particulier qui est dû aux caractéristiques de notre tâche migrante et qui affecte uniquement la Copie Directe et la *Prefetch Précopie*. Dans nos paramètres, il se trouve que cette taille mémoire correspond à un transfert très exactement égal à la durée d'inactivité de la tâche migrante i.e. $(T_m.p - T_m.d)$. Ainsi la tâche système générée pour la Copie Directe dispose d'un temps d'exécution et d'une échéance relative identiques : ceci a pour conséquence que cette tâche n'est jamais préemptée, de par la garantie que nous construisons uniquement des ordonnancements faisables. Ceci a pour effet de ne (quasiment) pas perturber l'exécution des autres tâches, la tâche système étant indivisible et ininterruptible. Ceci explique également que la génération de 100 jeux de tâches faisables pour $s(T_m) = 250$ ko ait demandé à peu près deux fois plus de temps que pour les autres valeurs. En effet, imposer une telle exécution est assez restrictif dans la génération de jeux de tâches faisables. La *Prefetch Précopie* se retrouve également concernée dans une moindre mesure car dépendant de la date de demande de migration : si celle-ci est dans le scénario pire cas, la *Prefetch Précopie* se comporte comme la Copie Directe sinon dans tous les autres cas les transferts sont dispersés. Ceci se traduit sur la courbe par une diminution plus légère. Les autres techniques divisent toujours les transferts quelque soit la date de demande et ne sont donc pas concernées par ce phénomène.

4.7.3 Discussions sur l'évaluation

Pour une taille mémoire de 1 Mb de la tâche migrable, les techniques de migration proposées exhibent un surcoût relativement faible en termes de performance : l'augmentation moyenne du nombre de préemptions varie entre 0 and 14%. Ceci n'est pas seulement vrai pour $N_j = 2$ mais aussi pour un nombre plus important de travaux par tâche. La technique qui montre le moins de surcoût vis à vis du ratio de préemptions est aussi celle qui est la plus restrictive en terme de faisabilité. Il s'agit de la Copie Directe. C'est compréhensible car cette technique ne divise pas les transferts de migrations.

La *Prefetch Précopie* est légèrement moins efficace en terme de préemptions produites pour les mêmes contraintes d'ordonnançabilité. Les deux autres techniques sont moins restrictives pour la faisabilité et l'analyse d'ordonnançabilité mais sont également moins efficaces vis à vis de ce ratio de préemptions.

Cependant, la Copie Mixte exhibe de meilleurs résultats que la *Prefetch Postcopie*, et devrait être employée par défaut si l'ensemble des tâches du système est déjà très contraint (en terme de charge CPU/réseau). Néanmoins, la copie Mixte ne peut pas gérer des fréquences de migration de tâches trop élevées (tous les deux cycles max.). Ceci ne concerne que des hautes fréquences qui ne sont pas réalistes dans le contexte de systèmes temps-réel stricts pour la tolérance aux fautes. Pour des ensembles de tâches moins contraints en ressources, la Copie Directe devrait être employée si le système est ordonnançable. Notons que la Copie Directe est la technique la plus simple à implémenter.

Pour des applications embarquées, le surcoût en préemption est très peu influencé par la taille mémoire de la tâche migrante quelque soit la technique employée. Ce surcoût est linéaire en fonction de la taille mémoire, hormis le cas particulier pour certaines techniques où la durée de transfert est très exactement égale à la durée d'inactivité de la tâche.

4.8 Conclusion

Nous proposons de nouvelles techniques pour s'assurer de la migration de tâche dans des systèmes distribués temps-réel stricts : *Prefetch Précopie*, *Prefetch Postcopie* et Copie Mixte. Nous montrons qu'il est possible d'éviter le temps de gel durant la migration par la connaissance des inactivités de la tâche migrante. Cette information est disponible lors de la conception de systèmes temps-réel stricts qui nécessitent une description statique des contraintes temporelles de l'ensemble des tâches.

Pour garantir la faisabilité de ces techniques, nous formulons un problème de programmation par contraintes, ce qui permet d'exprimer les propriétés d'une tâche système, en charge d'effectuer la migration des tâches. Cette tâche supplémentaire peut alors être prise en compte dans l'analyse globale d'ordonnançabilité d'un système distribué temps-réel strict. Nos résultats montrent que la technique classique de Copie Directe (adaptée pour le contexte de systèmes temps-réel) et la *Prefetch Précopie* fournissent des conditions de faisabilité identiques au niveau du travail tandis que la *Prefetch Postcopie* et la Copie Mixte permettent plus de flexibilité pour effectuer une migration.

Au niveau de la tâche, la faisabilité de la Copie Directe est la plus restrictive avec la *Prefetch Précopie* suivie de la *Prefetch Postcopie* et la Copie Mixte. L'analyse de faisabilité de la Copie Mixte est similaire à celle de la *Prefetch Postcopie* à la différence des dates de début des travaux systèmes qui peuvent être fixées dès la fin de leurs travaux migrants respectifs (les temps d'exécution et les échéances restants les mêmes).

Nous montrons également comment intégrer les tâches système utilisées pour implémenter les techniques de migration avec des tests d'ordonnançabilité bien connus. Nous discutons des paramètres qui peuvent être réglés en-ligne pour minimiser le surcoût des techniques de migration. Finalement, nous effectuons une évaluation de l'efficacité des techniques de migration qui se base sur le nombre de préemptions qu'elles provoquent. Il s'agit d'un critère d'évaluation qui montre que le surcoût engendré est acceptable : en dessous de 10 % pour de faibles fréquences de migration. L'influence du nombre de travaux par tâche a été aussi étudié. La technique la plus contrainte pour l'ordonnançabilité, la Copie Directe, est aussi celle qui est la plus efficace en termes de préemptions supplémentaires produites. L'influence de la taille mémoire est très légère pour des applications embarquées. Les résultats montrent enfin que la Copie Mixte dispose de résultats proches comparés à la Copie Directe. Comme la Copie Mixte fournit un avantage sur ces conditions de faisabilité, il s'agit donc d'un compromis intéressant.

Pour la suite, nous voulons évaluer l'efficacité de nos techniques de migrations sur un critère autre que celui du nombre de préemptions. Nous envisageons la définition d'un critère basé sur le surcoût moyen du réseau pour identifier les politiques qui permettent un meilleur équilibre sur ce critère. De plus, nous prévoyons de caractériser l'impact des effets de cache dûs à la migration de tâche dans l'analyse de faisabilité. Les expérimentations pourraient également être étendues à des modèles de tâches plus généraux, avec dépendances entre travaux, potentiellement non-cyliques [MNLM10] ou communicantes [PGH98]. Nous planifions enfin l'implémentation de ces techniques sur un outil prototype pour la conception de systèmes distribués temps-réel critiques.

Chapitre 5

Minimisation des préemptions et migrations sur des architectures multicœurs

Sommaire

L'exécution de tâches temps-réel sur une architecture multicœur à mémoire partagée a fait l'objet de nombreux travaux. Leur implémentation pratique sur une plateforme réelle est peu étudiée, à part les travaux initiés par Brandenburg [BCA08]. Ces travaux se concentrent sur la famille d'ordonnanceurs Pfair [BCPV93] et sur des algorithmes partitionnés. Plusieurs constats sont effectués : plusieurs types de surcoûts affectent l'exécution réelle et amènent à des taux d'utilisation garantis pour l'ordonnançabilité bien inférieurs aux capacités des processeurs, notamment quand le nombre de tâches augmente. L'un de ces surcoûts est le changement de contexte lors de préemptions ou de migrations de tâche. Les algorithmes globaux, tels les algorithmes de la famille Pfair évoqués ci-dessus, sont connus pour en produire beaucoup.

Parmi les caractéristiques des ordonnanceurs globaux temps-réel pour les systèmes multiprocesseurs, telles que les architectures multicœurs, l'une d'elle est particulièrement importante : l'optimalité. Il s'agit d'une propriété garantissant qu'un ordonnanceur peut en théorie utiliser toutes les capacités des processeurs sans dépassement d'échéances. Un algorithme optimal au sens temps-réel garantit l'obtention d'un ordonnancement correct, sans dépassement d'échéance, pour tout ensemble faisable de tâches.

Nous présentons dans ce chapitre une approche pour diminuer les préemptions et les migrations de tâches dans les ordonnancements temps-réel globaux optimaux pour multiprocesseurs.

Contrairement à la plupart des approches, notre méthode se déroule en deux étapes, l'une hors-ligne pour placer temporellement les travaux sur les intervalles, l'autre en-ligne pour les exécuter dynamiquement à l'intérieur de chaque intervalle. Nous proposons une formulation par programmation linéaire et un ordonnanceur original présentant une faible complexité et produisant peu de préemptions et de migrations de tâches. Sur le plan expérimental, nous comparons notre approche avec d'autres algorithmes d'ordonnancement existants en utilisant un modèle de tâches commun, celui des tâches périodiques à échéances implicites. Les résultats illustrent la compétitivité de notre approche vis à vis du nombre de préemptions et de migrations de tâches engendrées.

Ces travaux ont fait l'objet d'un poster au GDR SOC/SIP'09 [MCDF09] et d'un article long à RTSS'10 [MSD10].

5.1 Problématiques

Les ordonnancements en-ligne optimaux sont donc efficaces du point de vue de l'utilisation théorique de la capacité du système, mais sont souvent critiqués sur leur complexité et sur le fait qu'ils produisent comparativement un nombre plus important de préemptions et de migrations de tâches. Par conséquent, ils sont en théorie efficaces mais ne sont pas nécessairement utilisables en pratique en prenant compte les coûts de préemptions et de migrations : ce coût correspond aux changements de contexte et également aux décisions d'ordonnancement.

Les ordonnanceurs non-optimaux (e.g. Global-EDF, ou tout algorithme à priorités fixes) engendrent généralement moins de préemptions et migrations de tâches que les optimaux. Cependant, ils ne garantissent pas le respect des contraintes temporelles au dessus d'une borne supérieure du taux d'utilisation [CFH+04], ou alors en utilisant une technique par augmentation de ressource (voir chapitre 2). Il est donc nécessaire d'utiliser plus de ressources pour obtenir des ordonnancements corrects pour le même ensemble de tâches. De plus, des conditions nécessaires et suffisantes pour prouver l'ordonnançabilité n'existent pas toujours. Enfin, les temps de repos des processeurs ne peuvent pas toujours être systématiquement évités, ainsi de précieuses ressources sont perdues. Récemment, certains travaux ont tenté d'améliorer les tests de faisabilité des ordonnanceurs non-optimaux [Bak05] car ils produisent en pratique moins de préemptions que les optimaux. Notre but, dans ce chapitre, est de montrer qu'il est également possible de construire des ordonnanceurs optimaux engendrant peu de préemptions et de migrations.

La tendance actuelle est de privilégier l'ordonnancement aux dates de début/fin des travaux (*job-boundary*) comme le font Bfair [ZMC03] et LLREF [CRJ06]. Ces approches ont été récemment étudiées pour formaliser ce concept, qui a donné le nom d'ordonnanceur *DP-fair* [LFS+10] (*Deadline Partitionning*) : la mise en oeuvre de l'ordonnancement est partitionnée temporellement selon les dates d'échéances des travaux. Notre travail suit la même idée. Nous allons montrer que l'allocation de tâches de nature cyclique revient à ordonnancer un ensemble fini de travaux sur des intervalles consécutifs. Il est alors possible d'utiliser la représentation d'ordonnancement par poids, proposée par Lemerre et al. [LDAVN08]. Cette représentation permet d'exprimer les contraintes temps-réel sous une forme linéaire. En résolvant ce système linéaire d'inégalités, on obtient un test exact de faisabilité (i.e. une condition nécessaire et suffisante pour l'ordonnançabilité) et la quantité de travail nécessaire pour ordonnancer dynamiquement dans chaque intervalle. Le système linéaire peut être résolu en utilisant l'algorithme du simplexe qui propose une solution arbitraire en l'absence d'une fonction économique. Afin de sélectionner une meilleure solution vis à vis du nombre de préemptions et migrations engendrées, nous proposons

une formulation en nombres entiers avec des fonctions économiques appropriées.

La plupart des ordonnanceurs sont entièrement statiques ou entièrement dynamiques. Les ordonnanceurs statiques produisent un ordonnancement fixe, ils ne peuvent donc pas gérer des tâches imprévues (par exemple des tâches non-critiques dirigées par les évènements avec des dates de début inconnues) et ne peuvent tirés bénéfice des aléas d'exécution, lorsque la durée d'un travail est inférieure à son WCET par exemple. Il est alors intéressant de diviser le travail de l'ordonnanceur pour diminuer sa complexité en-ligne tout en permettant un comportement dynamique, ce qui permet de réaliser un système plus robuste. D'un point de vue méthodologique également, il est acceptable d'investir un temps non négligeable en compilation dans un contexte embarqué, au moins à la fin du cycle de développement logiciel.

Nous considérons dans cette étude des tâches périodiques pour faciliter la comparaison avec d'autres ordonnanceurs optimaux et non-optimaux. Le fait de considérer des tâches périodiques indépendantes ne veut pas dire qu'il n'est pas possible de gérer les communications : par exemple, le problème de synchronisation peut être résolu en utilisant des mécanismes lock-free [HS08]. Comme nous le verrons en Section 5.4, notre travail peut être étendu à des modèles de tâches plus généraux comme le modèle de tâches OASIS [LDAV10].

Dans la prochaine section, nous présentons quelques rappels sur le temps-réel puis un état de l'art dans la section suivante. La Section 5.4 définit l'approche générale et en Section 5.5, nous décrivons notre modèle linéaire. Nous évaluons sa complexité en Section 5.6 puis présentons une évaluation de l'efficacité en terme de nombre de préemptions et migrations engendrées par notre approche comparée à d'autres algorithmes connus, en Section 5.7. Enfin, nous faisons la synthèse de ces travaux en Section 5.8 puis nous présentons les extensions et perspectives.

5.2 Rappels temps-réel

Dans un système temps-réel, une *tâche* est un ensemble de traitements exécutés de manière séquentielle. Soit Γ un ensemble de tâches et une tâche $T_i \in \Gamma$, celle-ci peut être composée de plusieurs *travaux* consécutifs j_i en nombre fini ou infini (dépendant de la classe de tâches considérée) : $T_i = \{j_i^k, \forall j \in \mathbb{N}^*\}$. Chacune des exécutions est alors considérée de manière individuelle sous forme d'instance : j_i^k correspond à la $k^{\text{ème}}$ instance d'une tâche T_i.

Caractéristiques d'un travail $\mathbf{j_i^k}$

- $j_i^k.r$ la date de réveil (*release time*),
- $j_i^k.e$ le temps d'exécution au pire cas (*worst case execution time*),
- $j_i^k.d$ l'échéance relative à $j_i.r$ (*deadline*).

Lors de l'exécution, on peut observer le comportement temporel de chacun des travaux, notamment :

- $j_i^k.s$ sa date début (*start time*),
- $j_i^k.f$ sa date de fin de traitement (*completion time*).
- $j_i^k.l$ sa laxité, telle que $j_i^k.l = j_i^k.d - t - j_i^k.e(t)$, où t représente la date courante ($j_i^k.r \leq t \leq j_i^k.d$) et $j_i^k.e(t)$ le temps d'exécution restant à la date t.

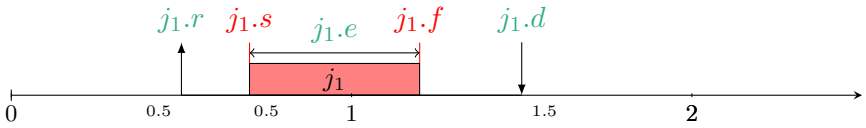


FIGURE 5.1 – Illustration des caractéristiques et observations d'un travail $j_1 = (0.5, 0.5, 1.5)$.

Si le travail n'est pas préempté, $j_i.f - j_i.s = j_i.e$: c'est toujours le cas si l'ordonnanceur est non-préemptif. Vérifier le dépassement d'échéances revient à vérifier sa laxité est négative i.e. $j_i.l < 0$.

Les théorèmes suivants établissent des résultats sur la non-optimalité multiprocesseur. Ils se basent sur une classe de tâches où les dates de réveil des tâches ne sont pas connues à l'avance.

Théorème 1. *Lorsqu'il y a 2 processeurs ou plus, il n'existe pas d'algorithme en-ligne permettant de construire un ordonnancement respectant toutes les échéances [Sah79].*

Théorème 2. *Pour plus de deux processeurs, il ne peut y avoir d'algorithme optimal (basé sur l'échéance) sans connaissance à priori des dates de début, temps d'exécution et échéances des prochains travaux [DM89].*

Théorème 3. *S'il y a plus de 2 processeurs, aucun algorithme d'ordonnancement optimal en-ligne ne peut exister pour des ensembles de tâches avec 2 échéances distinctes ou plus [HL92b].*

On comprend donc la nécessité de connaître au mieux les caractéristiques temporelles des tâches. Ceci explique pourquoi l'état l'art des ordonnanceurs multiprocesseur temps-réel optimaux se limite souvent à l'étude d'un modèle de tâches périodiques et/ou sporadiques qui permettent d'obtenir l'optimalité.

5.3 Positionnement

Dans les sections à venir, nous utiliserons les notations suivantes : on considère un ensemble Γ de N tâches s'exécutant sur M processeurs, chaque tâche T délivre des travaux périodiquement tous les $T.p$ unités de temps. Nous utilisons la terminologie suivante : soit H_Γ l'*hyperpériode* définie comme le plus petit commun multiple de l'ensemble des périodes des tâches et G_Γ défini comme le plus grand dénominateur de l'ensemble des périodes de tâches.

Les ordonnanceurs de la famille Pfair (PF [BCPV96], PD [BGP95], PD^2 [AS00]) obtiennent des ordonnancements optimaux pour les tâches temps-réel périodiques sur des architectures multiprocesseurs. Ces algorithmes se basent sur la notion d'équité proportionnée (*proportionate fairness*) qui essaye de suivre de manière proche un ordonnancement fluide (taux d'utilisation quasi constant) par un comportement dynamique et une approche basée sur des fenêtres : chaque tâche est décomposée en sous-tâches disposant de pseudo-échéances. Par conséquent, tous ces ordonnanceurs peuvent générer beaucoup de préemptions, au plus $M * H_\Gamma$ par hyperpériode. Dans cette famille, PD^2 est connu pour être le plus efficace en terme de complexité ($O(\min(N, M \log N))$.
Khemka *et al.* proposent un ordonnancement statique optimal [KS97], qui est construit hors-ligne de manière incrémentale sur des intervalles de temps définis par les multiples de G_Γ. L'allocation des tâches est fait individuellement par période croissante. Cependant, la complexité et le nombre de préemptions de tâches dépend du ratio entre H_Γ et G_Γ.
Dans la famille des ordonnanceurs *DP-fair*, le plus simple d'entre eux a été réalisé initialement

par Coffman [Cof76], puis par Lee et al. [LCC94]. Il consiste à réserver à chacune des tâches un temps proportionnel à leur taux d'utilisation (rapport temps d'exécution sur échéance) sur tous les intervalles : ces intervalles sont définis par les dates de début et de fin des travaux. Notons que cette approche a été reprise récemment sous le nom de *DP-Wrap* par Levin et al. [LFS+10] à la seule différence qu'ils proposent une extension pour la gestion des tâches sporadiques. Cependant, cet ordonnanceur oblige toutes les tâches à s'exécuter sur chacun des intervalles, ce qui peut générer un nombre de préemptions important par intervalle. LLREF [CRJ06] est un autre algorithme optimal qui prend dynamiquement des décisions d'ordonnancement sur des intervalles locaux définis entre deux apparitions de travaux. Deux évènements possibles sont gérés : soit un travail se termine soit un travail obtient une priorité haute (correspondant à un travail avec une laxité nulle). Les travaux avec les plus grands temps d'exécution restants s'exécutent prioritairement jusqu'à ce qu'un des deux évènements se produisent. Une borne supérieure de complexité de l'ordonnanceur est $O(N)$.
EKG [AT06] est un autre algorithme basé sur EDF [LL73] permettant une division des tâches par groupe de K processeurs. Il peut être utilisé comme un ordonnancement optimal s'il n'y a pas de groupe ($K = M$, les tâches migrent sur tous les processeurs) ou comme ordonnancement partitionné (les tâches ne peuvent alors migrer que dans leur groupe), mais alors avec un taux d'utilisation processeurs limité. Il engendre un nombre de préemptions et de migrations fonction du nombre de *travaux exécutés*, au plus $2 * K * N'$ où $N' = \sum_{T \in \Gamma}^{N} \frac{H_\Gamma}{T.p}$.

Améliorations sur le nombre de changements de contexte

Certains travaux ont essayé d'améliorer les algorithmes optimaux existants concernant le nombre de préemptions et migrations engendrées.
Zhu *et al.* ont proposé Bfair (Boundary fair [ZMC03]), une amélioration de Pfair, qui applique un ordonnancement fluide uniquement aux instants d'apparition des travaux (*job boundaries*). Il fait parti de la famille des ordonnanceurs *DP-Fair*. À chaque début d'intervalle, une approche basée sur Pfair choisit quels travaux seront exécutés sur l'intervalle puis l'algorithme de McNaughton [McN59] construit le plan d'ordonnancement statiquement : le nombre de préemptions est alors borné par $M - 1$ sur chacun des intervalles. On peut considérer cet ordonnancement de pseudo dynamique (car l'exécution est statique entre deux instants d'apparition de travaux) et sa complexité est en $O(N)$. Les auteurs montrent à partir de résultats expérimentaux que les préemptions de tâches sont réduites de 25 à 50 % comparé à Pfair.
Il est également possible d'améliorer le nombre de migrations de tâches en ajoutant des heuristiques à Pfair. Aoun *et al.* [ADT08] ont proposé quelques heuristiques pour guider l'allocation des sous-tâches pour garder l'exécution d'une tâche sur le même processeur autant que possible ou alors en réduisant la migration de sous-tâches d'un même travail. Leurs résultats expérimentaux montrent que le nombre total de migrations de tâches est réduit de 40 à 60% comparé à Pfair.
Funaoka *et al.* [FKY08] ont montré qu'il est possible de réduire significativement le nombre de préemptions de tâches avec un algorithme basé sur LLREF. Cela est faisable en utilisant localement un ordonnanceur non-oisif (i.e. l'algorithme ne laisse jamais les processeurs au repos s'il existe au moins une tâche en attente d'exécution dans le système) : lorsqu'un temps inutilisé existe sur un intervalle, l'algorithme essaye de le remplir complètement. Intuitivement, si un ensemble de tâches utilise moins que la capacité totale des processeurs, alors un ordonnanceur non-oisif permettra moins de divisions de travaux et ainsi engendrera moins de préemptions.
Le tableau 5.1 synthétise les caractéristiques théoriques d'un certain nombre d'ordonnanceurs globaux optimaux. Nous utilisons le fait qu'il existe au plus $1+\sum_{T \in \Gamma} \frac{H_\Gamma}{T.p}$ intervalles pendant une hyperpériode pour évaluer le nombre maximum de préemptions de tâches sur les ordonnanceurs basés sur des intervalles. En fait, une borne supérieure est de $\sum_{T \in \Gamma} \frac{H_\Gamma}{T.p}$ si toutes les tâches

Nom de l'algorithme	Pfair [BCPV96] [BGP95] [AS00]	Bfair [ZMC03]	DP-Wrap [LCC94] [LFS+10]	LLREF [CRJ06] [FKY08]	SA [KS97]	EKG [AT06]
Type d'allocation	dynamique	pseudo dynamique	dynamique	dynamique	statique	dynamique
Complexité hors-ligne	-	-	-	-	O(N∗ H_Γ/G_Γ)	-
Complexité en-ligne	O(min(N, Mlog(N))[AS00]	O(N)	O(N)	O(M)[FKY08] O(N)[CRJ06]	O(1)	O(N)
Borne sup. de préemptions par intervalle	-	M-1	M-1	N	-	-
Borne sup. de préemptions sur l'hyperpériode	M∗H_Γ	(M-1)∗ (1+ $\sum_{T\in\Gamma}\frac{H_\Gamma}{T.p}$)	(M-1)∗ (1+ $\sum_{T\in\Gamma}\frac{H_\Gamma}{T.p}$)	N∗ (1+ $\sum_{T\in\Gamma}\frac{H_\Gamma}{T.p}$)	(M+N-1)∗ H_Γ/G_Γ	2M∗ $\sum_{T\in\Gamma}\frac{H_\Gamma}{T.p}$
Type de processeur	identique	identique	identique	identique	uniforme	identique

TABLE 5.1 – Comparatif des caractéristiques d'un certain nombre d'ordonnanceurs globaux optimaux

sont synchrones (sans offset au démarrage). Nous comparons le type d'allocation, la complexité hors-ligne et en-ligne (si c'est pertinent), les bornes supérieures du nombre de préemptions par intervalle et sur l'hyperpériode pour un jeu de tâches périodiques utilisant toutes les capacités processeurs, et enfin le type de processeur sur lequel ils sont utilisés.

5.4 Approche générale

Dans cette section, nous présentons les définitions nécessaires pour comprendre les représentations d'ordonnancement et leur équivalence. Plus précisément dans notre contexte, nous cherchons à exécuter un ensemble fini de travaux. Nous considérons le fait qu'ordonnancer d'un ensemble de tâches périodiques est équivalent à ordonnancer un ensemble fini de travaux sur différents intervalles (de longueurs différentes) définis par les instants de début des travaux et exécutés de manière cyclique à chaque *hyperpériode*.

Nous utilisons le fait que la représentation de l'ordonnancement par poids et par intervalle (Boundary Weighted Schedule $\mathcal{BWS}$) est adapté à notre problème. Nous exprimons alors le système linéaire correspondant Σ pour l'approche générale et nous montrons comment le résoudre et exploiter le résultat afin de réaliser l'ordonnancement.

Nous rappelons ici quelques définitions et résultats basés sur les travaux publiés dans [LDAVN08]. Nous considérons un ensemble de tâches Γ défini par N tâches périodiques à échéances implicites.

Soit un ensemble de tâches Γ, chacune d'entre elles (T) produit périodiquement $(T.p)$ des travaux, chacun des travaux j se définit par son temps d'exécution $j.e$ et son échéance $j.d$.

Tous les travaux exécutés pendant l'*hyperpériode* H_Γ appartiennent à l'ensemble des travaux J_Γ. Comme nous cherchons à diviser H_Γ en plusieurs intervalles, nous définissons J_k comme le sous-ensemble de travaux de J_Γ qui contient tous les travaux actifs (potentiellement exécutés) sur le $k^{\text{ème}}$ intervalle (la notion d'intervalle est définie comme pour les autres algorithmes *DP-Wrap*, pour une définition précise, voir ci-dessous).

Soit J un ensemble de travaux, une représentation d'ordonnancement associe deux objets : l'ensemble des ordonnancements corrects S_J et une fonction d'exécution qui représente le temps passé à exécuter J. On distingue habituellement quatre représentations d'ordonnancement :

- $\mathcal{CS}$: la représentation concrète (qui indique lorsqu'un travail est exécuté et sur quel processeur, i.e. c'est un diagramme de Gantt),
- $\mathcal{AS}$: la représentation anonyme (qui indique quand un travail est exécuté sur un processeur),
- $\mathcal{WS}$: la représentation par poids (qui indique quelle fraction processeur est allouée à quel travail et à quel moment),
- $\mathcal{BWS}$: la représentation par poids et par intervalle (qui indique quelle fraction processeur est allouée à quel travail, avec comme restriction que ces fractions ne changent seulement qu'au début d'un intervalle).

Théorème 4. *$\mathcal{CS}$, $\mathcal{AS}$, $\mathcal{WS}$ et $\mathcal{BWS}$ sont quatre représentations équivalentes.*

Le lecteur peut se référer à [LDAVN08] pour les preuves et pour les illustrations correspondantes à chaque représentation.

Pour trouver un ordonnancement complet, il suffit d'exécuter l'ensemble des tâches sur l'*hyperpériode*. En considérant tous les travaux de toutes les tâches J_Γ, il est possible de diviser la durée de l'hyperpériode en un nombre fini et consécutif d'intervalles définis par les dates de début des travaux (voir l'exemple de la Fig. 5.2). Soit I l'ensemble des intervalles et I_k le $k^{\text{ème}}$ intervalle, nous définissons la durée $|I_k|$ l'intervalle de temps entre la date t_k et la date t_{k+1} (correspondant aux $k^{\text{ème}}$ et $(k+1)^{\text{ème}}$ instants non commun de début des travaux de J_Γ) :

$$|I_k| = t_{k+1} - t_k$$

Dans notre contexte, nous connaissons toujours l'instant correspondant à la prochaine date de début i.e. lorsque la date t_k est atteinte alors t_{k+1} est connue. C'est le cas pour le modèle de tâches périodiques, et plus généralement pour des modèles *time-triggered* comme les automates temporels [LDAV10].

Le poids w_j d'un travail $j \in J$ se définit comme la fraction de processeur nécessaire pour exécuter le travail pendant sa durée d'exécution entre sa date de début et son échéance. Étant donné qu'un travail j peut être présent sur plusieurs intervalles (i.e. un travail peut se diviser en plusieurs sous-travaux), nous notons $w_{j,k}$ le poids du sous-travail de j sur l'intervalle I_k. Comme pour les travaux, le poids d'un sous-travail est défini comme la quantité de capacité processeur nécessaire pour exécuter le travail j mais seulement sur l'intervalle I_k.

L'ensemble E_j correspond à l'ensemble des numéros d'intervalle sur lesquels les sous-travaux d'un travail j peuvent s'exécuter : cela peut correspondre à un ou plusieurs intervalles. Par exemple, le travail j_{22} sur la Fig. 5.2-a peut s'exécuter sur l'intervalle I_2 et I_3, donc $E_{j_{22}} = \{2, 3\}$.

Comme indiqué après, un problème d'ordonnancement d'un ensemble fini de travaux peut s'exprimer comme un simple problème de programmation linéaire [1].

5.4.1 Définition d'un système d'égalités et inégalités linéaires

Nous définissons notre système d'équations linéaires Σ qui permet :

- de vérifier l'ordonnançabilité d'un ensemble de travaux en multiprocesseurs (composé de M processeurs identiques).
- d'obtenir les poids des sous-travaux pour chaque intervalle, par la résolution de Σ (si et seulement si l'ensemble des travaux est ordonnançable).

Premièrement, nous exprimons les contraintes de validité :

$$\sum_{j \in J_k} w_{j,k} \leq M, \forall k \tag{5.1}$$

$$0 \leq w_{j,k} \leq 1, \forall j \forall k \tag{5.2}$$

La première inégalité (5.1) traduit le fait que la somme des poids des sous-travaux sur un intervalle ne doit pas dépasser la capacité maximale des processeurs. La deuxième (5.2) traduit le fait que chaque sous-travail ne doit pas dépasser la capacité maximale d'un processeur : c'est une condition nécessaire pour éviter qu'un même travail se voit exécuté sur plusieurs processeurs en même temps. Ensuite, nous avons l'ensemble des équations de cohérence,

$$\sum_{k \in E_j} w_{j,k} * |I_k| = j.e, \forall j \tag{5.3}$$

qui décrivent que la durée d'un travail est égale à la somme des temps d'exécution de ses sous-travaux correspondants, i.e. les travaux doivent s'exécuter en intégralité.

5.4.2 Approche de résolution

Résoudre le programme linéaire Σ donné par les Eqs. (5.1), (5.2) et (5.3) permet de trouver l'ensemble des poids $w_{j,k}$ si l'ensemble des tâches est *faisable*. Si c'est le cas, le problème d'ordonnancement se réduit à exécuter des sous-travaux sur des intervalles consécutifs. Sur chaque intervalle I_k, les sous-travaux correspondants ont la même date de début et la même échéance, définies par les limites des intervalles et leur temps d'exécution est donné par $w_{j,k} * |I_k|$. Sur l'exemple de la Fig. 5.2-a : considérons trois tâches périodiques T_1, T_2 et T_3 produisant des travaux périodiquement (toutes les 6, 9 et 18 unités de temps respectivement) de temps d'exécution 4, 6 et 12 respectivement. Le taux d'utilisation processeur est égal à $M = 2$. A titre d'exemple, nous pouvons écrire le système correspondant Σ_1 comme montré à la section précédente :

$$\Sigma_1 \left| \begin{array}{ll} w_{11,0} * 6 = 4 & w_{11,0} + w_{21,0} + w_{3,0} \leq 2 \\ w_{12,1} * 3 + w_{12,2} * 3 = 4 & w_{12,1} + w_{21,1} + w_{3,1} \leq 2 \\ w_{13,3} * 6 = 4 & w_{12,2} + w_{22,2} + w_{3,2} \leq 2 \\ w_{21,0} * 6 + w_{21,1} * 3 = 6 & w_{13,3} + w_{22,3} + w_{3,3} \leq 2 \\ w_{22,2} * 3 + w_{22,3} * 6 = 6 & \\ w_{3,0} * 6 + w_{3,1} * 3 + w_{3,2} * 3 + w_{3,3} * 6 = 12 & \\ w_{11,0}, w_{12,1}, w_{12,2}, w_{13,3}, w_{21,0}, w_{21,1} \in [0,1] & \\ w_{22,2}, w_{22,3}, w_{3,0}, w_{3,1}, w_{3,2}, w_{3,3} \in [0,1] & \end{array} \right.$$

1. Ce qui s'avère être un problème de flot, comme montré dans [BFR71].

en résolvant Σ_1, nous obtenons l'ensemble des douze poids pour les quatre intervalles correspondants (sur la Fig. 5.2-b apparaissent les poids non nuls). À partir de là, il devient possible d'appliquer un algorithme approprié pour allouer les sous travaux aux processeurs sur chacun des intervalles. Il existe différents ordonnanceurs pour cette classe de travaux (ayant même date de début et même échéance) tels que l'algorithme de McNaughton [McN59] (utilisé dans Bfair), celui de Gonzalez [GS78] ou plus récemment EDZL [WCL+07].

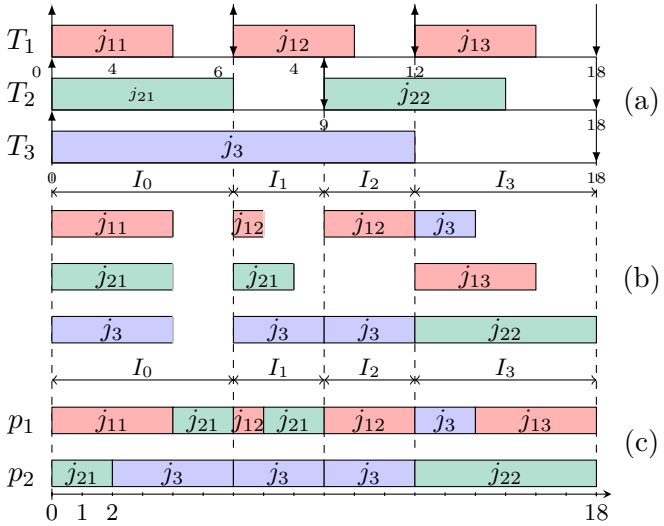


FIGURE 5.2 – Exemple avec trois tâches périodiques sur deux processeurs p_1, p_2 : (a) les tâches T_1, T_2 and T_3 délivrent des travaux périodiquement (toutes les 6, 9 and 18 unités respectivement) avec des temps d'exécution de 4, 6 and 12 unités de temps respectivement. (b) les durées des travaux sont triés par ordre croissant des temps d'exécution pour chaque intervalle et pour chaque travail résolvant le système Σ_1. (c) : un ordonnancement correspondant avec IZL.

Tous conviennent (car optimaux) mais nous proposons un nouvel algorithme plus approprié de par son caractère dynamique, sa faible complexité et de par le fait qu'il engendre peu de préemptions et migrations.

5.4.3 Description d'ordonnancement

Notre algorithme d'ordonnancement, IZL [2] (*Incremental scheduling with Zero Laxity*), est une variante d'un algorithme initialement proposé par Lemerre *et al.* (appelé *Algorithm 1* dans leur article [LDAVN08]). Cet algorithme utilise les structures de données suivantes :

- $\mathcal{S}$ est un *tableau* contenant l'ensemble des travaux en exécution. Si $\mathcal{S}_i$ est vide, alors le processeur i sera au repos.
- $\mathcal{P}$ est une *deque* [3] contenant les processeurs avec les sous-travaux non-urgents (i.e. ceux qui ont une laxité locale positive). Lorsqu'un processeur exécute un sous-travail ayant une laxité locale nulle, il est enlevé de la liste $\mathcal{P}$.
- $\mathcal{Q}$ est une *deque* contenant les travaux en attente, non terminés et triés par ordre croissant des temps d'exécution. Les travaux dans $\mathcal{Q}$ sont ceux ayant les temps d'exécution restants

2. IZL qui a déjà été présenté succinctement dans un article court du GDR SOPSIP'09 [MCDF09].
3. Une deque (*double-ended queue*) est une file dont les opérations d'ajouts et suppressions se font en début ou en fin de file, et en temps constant [Knu73].

les plus longs.

La fonction `schedule`($\mathcal{S}$, t) alloue les travaux donnés par le tableau $\mathcal{S}$ pour une durée t sur leur processeurs respectifs, et pour chaque travail j décrémente son *temps restant* $j.e$, de t. Cette fonction décrémente enfin le *temps global restant* R de t. Pour finir, les opérations `is_empty`, `pop_`$*$, `push_`$*$ sont des opérations standards sur les deque.

Algorithme IZL (Entrées : J_k, I_k) sur M processeurs identiques

```
1  Q←J_k (prérequis : les travaux sont triés par ordre croissant d'exécution)
2  R←|I_k|
3  S←(null, null, ..., null)
4  pour p from 1 to M faire
5      si is_empty(Q) alors faire
6          schedule(S, R)
7          retourne
8      fin si
9      S_p ← pop_first(Q)
10     push_last(p in P)
11 fin pour
12 tant que ¬is_empty(Q) faire
13     p_min ← first(P)
14     L ← last(Q)
15     si S_{p_min}.e ≥ R − L.e alors faire
16         schedule(S, R − L.e)
17         p_max ← pop_last(P)
18         push_first(S_{p_max} in Q)
19         S_{p_max} ← L
20         pop_last(Q)
21     fin si
22     sinon faire
23         schedule(S, S_{p_min}.e)
24         p_min ← pop_first(P)
25         S_{p_min} ← pop_first(Q)
26         push_last(p_min in P)
27     fin si
28 fin tant que
29 pour all p in P faire
30     schedule(S, S_p.e)
31     S_p ← null
32 fin pour
33 schedule(S, R)
```

Comme pour Lemerre *et al.*, IZL détermine la prochaine date de préemption ou migration uniquement à la précédente préemption/migration. Cela consiste à surveiller les plus longs travaux de $\mathcal{Q}$ et à leur réserver des processeurs lorsqu'ils deviennent urgents (correspondant aux évènements de laxité locale nulle, voir j_3 à $t = 2$, Fig. 5.2-c). Cependant la différence majeure vient du fait que ces évènements influencent différemment le plan d'ordonnancement (*l.15-21*) : les processeurs qui leur sont réservés sont ceux qui exécutaient les plus longs travaux parmi ceux en exécution (ces travaux sont alors enlevés et réinsérés à la tête de $\mathcal{Q}$ pour libérer ces processeurs). Cette méthode nécessite moins d'opérations et de préemptions de travaux (d'un facteur 2 par rapport à l'*Algorithm 1* puisqu'un évènement urgent n'entraîne plus qu'une préemption et migration).

Nous devons maintenant identifier les invariants pour apporter la preuve de fonctionnement de cet algorithme. Chaque invariant fait l'objet d'un lemme. Voici les invariants que nous allons démontrer :

Lemme 3. *$\mathcal{Q}$ est triée par ordre croissant des durées des travaux, $\forall j_1, j_2 \in \mathcal{Q}$:*

$$j_1 \text{ placé avant } j_2 \text{ in } \mathcal{Q} \Leftrightarrow j_1.e \leq j_2.e$$

Démonstration. Cet invariant est vrai au départ ($\mathcal{Q} = J_k$). Ensuite, seulement deux opérations sont possibles sur $\mathcal{Q}$:

- Opération d'extraction : on extrait le premier travail de $\mathcal{Q}$ lors de la première boucle **for** (*l.9*) et lorsqu'un travail se termine (*l.25*). Cependant, nous extrayons uniquement le dernier travail de $\mathcal{Q}$ lorsque sa laxité est nulle (*l.20*). Les éléments restants de la deque sont donc toujours triés.
- Opération d'addition : on ajoute uniquement un travail qui était en exécution au début de $\mathcal{Q}$ (*l.18*). Ce travail a été choisi pour être exécuté car son temps d'exécution était inférieur à ceux restants dans $\mathcal{Q}$ et comme il a été exécuté, son temps d'exécution n'a pu que décroître (tandis que ceux de $\mathcal{Q}$ n'ont pas évolué). Donc l'ajout d'un travail en exécution à la tête de la deque laisse la deque triée.

□

Lemme 4. *$\mathcal{P}$ sort les processeurs dans l'ordre de durée croissante des travaux qu'ils exécutent, $\forall p_1, p_2 \in \mathcal{P}$:*

$$p_1 \text{ placé avant } p_2 \text{ in } \mathcal{P} \Leftrightarrow \mathcal{S}_{p_1}.e \leq \mathcal{S}_{p_2}.e$$

Démonstration. $\mathcal{P}$ est toujours triée à la vue des trois opérations possibles la concernant :
a) dans la boucle **for** allouant les processeurs aux travaux triés, b) dans la branche **if**, on extrait p_{max} le dernier processeur de $\mathcal{P}$, c) dans la branche **else** p_{min} exécute un nouveau travail donc p_{min} est enlevé et remis à la fin de $\mathcal{P}$. □

Lemme 5. *Un travail ne peut apparaître qu'au plus une fois dans $\mathcal{S} \cup \mathcal{Q}$*

Démonstration. Cette invariant traduit le fait qu'à chaque fois qu'un élément est extrait de $\mathcal{Q}$, cela correspond à une allocation (S_p) : *l.9*, *l.14* puis *l.19-20* et *l.25*. □

Lemme 6. *Les travaux sont ordonnancés sur $\mathcal{P}$ et ont les temps d'exécution restants non nuls les plus petits :*

$$\forall j_S \in \mathcal{S}_\mathcal{P},\ j_Q \in \mathcal{Q},\ j_S.e \leq j_Q.e$$

Démonstration. Dans la boucle **for** (*l.4*) les travaux les plus courts sont extraits. Dans la boucle **while**, deux cas se présentent : soit le travail se termine donc il est remplacé par le premier des travaux restants de $\mathcal{Q}$ (*l.23-26*) soit le travail le plus long en exécution est remplacé par un autre urgent et le processeur correspondant est alors extrait de $\mathcal{P}$. Les travaux exécutés sur $\mathcal{P}$ sont donc toujours les plus courts en terme de temps d'exécution restants. □

Lemme 7. *A tout moment, R est le temps d'exécution restant pour arriver à la fin de la durée $|I_k|$*

Démonstration. *R* est initialisé pour la durée de l'intervalle et décroît à chaque appel de `schedule` (*l.6*, *l.16*, *l.23*, *l.30*, *l.33*).

□

Lemme 8. *Les travaux respectent toujours les conditions d'ordonnancement multiprocesseurs :*

$$\forall j \in J_k, j.e \leq R \text{ (a) } \& \sum_{j \in J_k} j.e \leq M * R \text{ (b)}$$

Démonstration. Au début de l'algorithme, les conditions (a) et (b) sont vérifiés par construction, voir Eqs. (5.1) et (5.2). Nous définissons $t = min(R - L.e, \mathcal{S}_p.e)$ et $L = \texttt{last}(\mathcal{Q})$.
(a) : Grâce aux invariants des lemmes 1, 4 et 5 nous savons que,

$$L.e \leq R - (R - L.e) \Rightarrow L.e \leq t$$

$$\forall j \in \mathcal{S} \cup \mathcal{Q},\ j.e \leq L.e \Rightarrow j.e \leq L.e \leq R - t$$

La condition d'ordonnançabilité (a) est donc vérifiée.

(b) : divisons cette preuve en deux parties. Premièrement, on considère le cas $|J_k| \leq M$. Alors $\mathcal{Q}$ sera vide soit avant la fin de la boucle **for** (ceci est traité *l.5-8*) ou à la fin de la seconde boucle **for** (*l.29-32*). Tous les travaux sont en exécution, la condition (b) est donc vérifiée. Deuxièmement, on envisage le cas où $|J_k| > M$. Nous soustrayons toujours la même quantité des deux côtés de l'inégalité (b) seulement lorsque `schedule` est appelé : à chaque itération de la boucle **while** (soit dans la branche **if** ou dans la branche **else**) et à la fin de l'algorithme (*l.30* et *l.33*). Cette quantité est de $M * t$ dans la boucle **while**, de $M * \mathcal{S}_p.e$ dans la seconde boucle **for** et de R (à la fin). L'inégalité (b) est donc toujours vérifiée. □

Maintenant que les invariants sont prouvés, il reste à montrer que l'algorithme se termine quelque soit le cas envisagé (à cause de la présence de la boucle **while**).

Lemme 9. *L'algorithme IZL termine quelque soit le cas envisagé.*

Démonstration. Pour ce faire, nous allons en fait prouver que le couple $(|\mathcal{P}|, |\mathcal{Q}|)$ est strictement décroissant :

- Si la branche **if** est prise dans la boucle **while**, ce couple devient $(|\mathcal{P}| - 1, |\mathcal{Q}|)$.
- Si la branche **else** est prise dans la boucle **while**, ce couple devient $(|\mathcal{P}|, |\mathcal{Q}| - 1)$.

Le cas $(|\mathcal{P}| = 0, |\mathcal{Q}| > 0)$ n'est pas possible, car il impliquerait que l'ensemble des travaux n'était pas ordonnançable. Ceci est donc faux par hypothèse. On en déduit la propriété suivante :

$$|\mathcal{P}| = 0 \Rightarrow |\mathcal{Q}| = 0$$

qui signifie que si tous les processeurs exécutent des travaux urgents alors il n'y a plus d'autres travaux en attente. On en déduit également que le couple $(|\mathcal{P}|, |\mathcal{Q}|)$ est donc strictement décroissant. □

Théorème 5. *L'algorithme IZL est optimal pour la classe de tâches ayant même début même fin.*

Démonstration. De par les lemmes 8 et 9, une condition nécessaire et suffisante est démontrée prouvant qu'IZL garantit une exécution correcte et qu'il assure la terminaison de toutes les tâches, même si le taux d'utilisation est maximal ($U = M$). □

Propriétés

- l'ordonnancement est construit de manière incrémentale ce qui signifie qu'il est *non-oisif* et *dynamique* sur l'intervalle.
- l'algorithme dispose d'un très faible complexité de calcul : $O(M)$ au tout début de l'intervalle (*l.4-11*) puis $O(1)$ pour tous les autres appels à l'ordonnanceur.

- l'ordonnanceur génère au plus $M-1$ préemptions et migrations par intervalle : aussi compétitif que celui de McNaughton's, connu pour être le meilleur algorithme statique de cette classe de tâches (voir Table 5.2). Pour le démontrer, il suffit de constater que les migrations et préemptions ne se produisent que lors des évènements urgents (un travail de $\mathcal{Q}$ atteint une laxité nulle) : pour être précis, il se produit une préemption et une migration (voir Fig 5.2-c à $t=2$). À chaque fois, un processeur est enlevé de $\mathcal{P}$. Lorsque $M-1$ processeurs sont enlevés, l'apparition d'un nouvel évènement urgent n'entraînera ni migration (car tous les autres processeurs sont déjà occupés avec un travail urgent) ni préemption (car comme nous l'avons vu $|\mathcal{P}|=0$ implique que tous les travaux sont exécutés). Il y a donc bien au plus $M-1$ préemptions et migrations.

Nom de l'algorithme	Préemptions	Complexité en-ligne	Type d'allocation
McNaugthon [McN59]	$\leq M-1$	$O(1)$	statique
Level [HLS77]	$\leq N^2(M-1)$	$O(MN^2)$	dynamique
SCH [GS78]	$\leq M-1$	$O(N+MlogM)$	dynamique
Algorithm 1 [LDAVN08]	$\leq 2M-2$	$O(1)$	dynamique
IZL	$\leq M-1$	$O(M)$ puis $O(1)$	dynamique

TABLE 5.2 – Comparatif des ordonnanceurs de la classe de tâches ayant même début, même fin sur des processeurs identiques, avec notre ordonnanceur IZL (*Incremental with Zero Laxity*).

Dans cette section nous avons introduit plusieurs notions. Nous avons vu comment exprimer les contraintes temps-réel par des équations et inéquations linéaires (le système Σ construit à partir des Eqs. (5.1), (5.2) et (5.3)). Nous avons utilisé le concept d'ordonnancement par poids et par intervalle et son équivalence avec la représentation concrète. Le programme linéaire peut être résolu avec l'algorithme du simplexe par exemple. Lorsqu'une solution est trouvée, le problème d'ordonnancement est réduit à l'exécution de sous-travaux avec les mêmes dates d'activation et d'échéance. Nous avons proposé un algorithme efficace (IZL) pour ordonnancer ces sous-travaux. Cependant, un programme linéaire sans une fonction économique produit des solutions faisables arbitraires ce qui n'est pas forcément approprié pour limiter les préemptions et migrations (voir Fig. 5.3-e,f). C'est la raison pour laquelle nous proposons d'améliorer cette approche en prenant en compte explicitement les préemptions de tâches.

5.5 Optimisation à l'aide d'un modèle linéaire en nombres entiers

Dans cette section, nous proposons un modèle linéaire en nombres entiers approprié afin de guider la solution sur les sommets du polyèdre qui impliquent moins de préemptions et de migrations de tâche. L'idée principale est d'ajouter des contraintes à notre système linéaire ainsi qu'une fonction économique qui vise à minimiser le nombre de préemptions de travaux (et par conséquence les migrations). Il est important de noter que le fait d'ajouter ces contraintes ne modifie pas l'ensemble des sommets de la projection du polyèdre sur l'espace vectoriel des variables initiales.

5.5.1 Amélioration employant des contraintes en supplément

Dans un programme linéaire mixte (MILP), certaines variables sont entières et les autres sont réelles. Notre modèle utilise des variables booléennes et entières : le programme linéaire (LP)

sera donc transformé en un MILP. Pour traduire notre besoin de minimiser les préemptions de tâches, une approche est de minimiser le nombre de sous-travaux présent sur chaque intervalle (à titre de comparaison, l'approche DP-Wrap [LFS+10] oblige tous les travaux à être présents sur chacun des intervalles).

Pour savoir si un travail est présent sur un intervalle, nous utilisons une variable booléenne : si son poids correspondant est strictement positif, le travail sera exécuté, sinon il ne sera pas considéré. Soit $x_{j,k}$, un booléen qui indique la présence d'un travail sur un intervalle I_k. Nous l'écrivons sous forme d'une contraintes comme indiqué :

$$x_{j,k} \geq w_{j,k} \tag{5.4}$$

Remarquons que cette variable fournit une condition nécessaire mais pas forcément suffisante de la présence d'un travail sur un intervalle. Nous adresserons ce problème plus tard dans ce chapitre.

Afin de compléter notre méthode, nous définissons, la préemption d'un travail lorsqu'un travail j est divisé en plusieurs sous-travaux (i.e. $|E_j| > 1$) lors de la résolution du système linéaire. Une préemption apparaîtra nécessairement si le poids du sous-travail est positif sur un intervalle et devient nul sur l'intervalle suivant.

Soit $y_{j,k}$ un booléen à l'état *vrai* si une préemption se produit nécessairement entre deux intervalles I_k and I_{k+1}. Nous définissons $y_{j,k}$ tel que :

$$y_{j,k} = \begin{cases} 1 & \text{si } x_{j,k} = 1 \text{ et } x_{j,k+1} = 0 \\ 0 & \text{sinon} \end{cases} \tag{5.5}$$

Remarquons que cela reviendrait au même de définir la préemption comme le passage d'un poids nul à un poids positif entre deux intervalles ($y_{j,k}$ à 1 si $x_{j,k} = 0$ et $x_{j,k+1} = 1$).
Pour chaque $y_{j,k}$, cela peut être modéliser en utilisant une contrainte quadratique telle que :

$$y_{j,k} = x_{j,k}(1 - x_{j,k+1}) \tag{5.6}$$

Il est tout de même possible de linéariser l'équation ci-dessus (5.6) en la remplaçant par l'ensemble de contraintes suivant [Pad78].

$$(5.6) \Leftrightarrow \begin{cases} x_{j,k} - x_{j,k+1} - y_{j,k} \leq 0 \\ -x_{j,k} + y_{j,k} \leq 0 \\ x_{j,k+1} + y_{j,k} \leq 1 \\ -y_{j,k} \leq 0 \end{cases}$$

Avec ces contraintes additionnelles, nous sommes capables de connaître quand va se produire une préemption et entre quels intervalles. Remarquons que pour tout travail, il y a une variable de préemption de moins que de variables de présence, puisque la préemption nécessite l'utilisation de deux variables consécutives. Dans le but de minimiser le nombre de fois que cela se produit, nous proposons d'ajouter ces contraintes au système Σ.

Un programme linéaire sans une fonction économique produit des solutions faisables arbitraires, nous proposons donc plusieurs fonctions économiques. Un premier paramètre que nous allons pouvoir utiliser est la minimisation du nombre de préemptions de travail pour borner

l'ensemble des préemptions de tous les travaux. Nous introduisons donc une variable entière y_j pour compter le nombre de préemptions par travail :

$$y_j = \sum_{k \in E_j \setminus \{\max_j E_j\}} y_{j,k}, \forall j \tag{5.7}$$

Il reste à exprimer la manière d'influencer ce nombre de préemptions de travail. Quatre possibilités sont considérées.

1) *Minimisation du nombre maximum de préemptions.*
Cela consiste à compter le nombre de préemptions pour chacun des travaux de chaque tâche et à minimiser la valeur maximale.

$$\text{minimiser } \max_j y_j \text{ avec } y_j \in \mathbb{N} \tag{5.8}$$

Cette expression peut être linéarisée de cette manière :

$$\begin{cases} \forall j, b_1 \geq y_j \text{ avec } b_1 \in \mathbb{N} \\ \text{minimiser } b_1 \end{cases}$$

2) *Minimisation du nombre total de préemptions.*
Cette deuxième fonction consiste à sommer les préemptions de tous les travaux de toutes les tâches et de minimiser la somme.

$$\text{minimiser } \sum_j y_j \tag{5.9}$$

3) *Minimisation du nombre total de présences des travaux.*
Cette troisième fonction économique traduit le besoin de minimiser la présence des sous-travaux : si nous autorisons la présence d'un moins grand nombre de sous-travaux nous minimisons également le nombre de préemptions de travaux entre intervalles. Nous avons tout d'abord besoin de compter le nombre de sous-travaux non-nuls pour chaque travail, donc nous introduisons une nouvelle variable entière x_j :

$$\sum_{k \in E_j} x_{j,k} = x_j, \forall j \tag{5.10}$$

Il reste à minimiser la somme de tous les x_j :

$$\text{minimiser } \sum_j x_j \tag{5.11}$$

4) *Minimisation du nombre total de préemptions et de présences des travaux.*
Cette dernière méthode consiste à sommer les préemptions et le nombre de présences de tous les travaux (ou des sous-travaux potentiels) de toutes les tâches et de minimiser le somme des deux. Nous exprimons la fonction économique à l'aide des Eqs. (5.7) et (5.10) :

$$\text{minimiser } \sum_j x_j + y_j \tag{5.12}$$

Limite nécessaire pour les $\mathbf{x_{j,k}}$ Une solution simple pour résoudre notre modèle entier est d'assigner 1 à toutes les variables $x_{j,k}$, quelque soit le critère de minimisation choisi (nous en proposons quatre). Cependant, les $w_{j,k}$ ne seront pas nécessairement strictement positifs. Pour éviter cela et résoudre ce problème évoqué en *remarque 1*, l'approche suivante est proposée.

Maximisation des poids des sous-travaux
Cette technique vise à maximiser les poids des sous-travaux uniquement lorsqu'ils sont présents sur l'intervalle ($x_{j,k} = 1$). Nous utilisons pour cela une variable réelle : $\alpha \in [0;1]$. Le but est de trouver la valeur maximale de α qui garde une solution valide : plus α est proche de 1, moins les travaux seront divisés. Nous procédons par dichotomie pour trouver la valeur critique. Soit $\alpha \in [0,1]$,

$$\begin{cases} \forall j, \forall k, w_{j,k} * |I_k| \geq \min(\alpha * j.e, |I_k|) * x_{j,k} \\ \text{maximiser } \alpha \end{cases} \tag{5.13}$$

La valeur de $\alpha * j.e$ pourrait dépasser $|I_k|$ si α est trop grand ce qui se traduirait par l'absence de solution (l'Eq. (5.2) entrerait en contradiction avec l'Eq. (5.13)), nous utilisons donc une fonction `min`() pour borner cette valeur.

Nous allons maintenant résumer les quatre programmes linéaires correspondants à nos quatre fonctions économiques :

$$\begin{cases} \text{Minimiser } b_{1/2/3/4} \text{ avec } \begin{cases} b_1 \geq y_j, b_1 \in \mathbb{N}, \forall j \\ b_2 = \sum_j y_j, b_2 \in \mathbb{N} \\ b_3 = \sum_j x_j, b_3 \in \mathbb{N} \\ b_4 = \sum_j x_j + y_j, b_4 \in \mathbb{N} \end{cases} \\ \text{s. l. c.} \\ \forall j,\ x_j = \sum_{k \in E_j} x_{j,k} \text{ et } y_j = \sum_{k \in E_j \setminus \{max_j E_j\}} y_{j,k} \\ \forall y_{j,k}, \begin{cases} x_{j,k} - x_{j,k+1} - y_{j,k} \leq 0 \\ -x_{j,k} + y_{j,k} \leq 0 \\ x_{j,k+1} + y_{j,k} \leq 1 \\ -y_{j,k} \leq 0 \end{cases} \\ \forall j, \forall k, \begin{cases} x_{j,k} \geq w_{j,k} \\ w_{j,k} * |I_k| \geq \alpha * j.e * x_{j,k},\ \alpha \in [0,1] \end{cases} \\ \forall j, \begin{cases} \sum_{j \in J_k} w_{j,k} \leq M, \forall k \\ 0 \leq w_{j,k} \leq 1, \forall k \\ \sum_{k \in E_j} w_{j,k} * |I_k| = j.e \end{cases} \end{cases} \tag{5.14}$$

Tous ces critères forment notre modèle mixte : les quatre programmes linéaires doivent être considérés séparément (les objectifs b_1, b_2, b_3 et b_4 correspondent aux Eqs. (5.8), (5.9), (5.11) et (5.12)) i.e. nous disposons exactement de quatre fonctions économiques distinctes.
Nous proposons plusieurs façons de minimiser les préemptions de travaux car ces différents critères peuvent aboutir à des résultats différents et parfois, l'un d'eux peut être plus approprié pour un ensemble de tâches particulier. Afin d'illustrer notre approche, nous présentons dans la section suivante les résultats d'ordonnancement pour l'une de ces fonctions économiques, en comparaison d'autres ordonnanceurs globaux optimaux.

5.5.2 Exemple d'ordonnancement

On considère l'exemple suivant initialement proposé par Zhu *et al.* [ZMC03] pour comparer Pfair et Bfair : six tâches périodiques à échéances implicites ($T.e$,$T.p$) : T_1=(2,5), T_2=(3,15), T_3=(3,15), T_4=(2,6), T_5=(20,30), T_6=(6,30). L'utilisation système est de $\sum_{T \in \Gamma}^{6} \frac{T.e}{T.p} = 2$ et l'hyperpériode de $H_\Gamma = 30$.
Remarquons par exemple que notre approche (Fig. 5.3-f) ne divise pas la tâche T_5 qui elle la

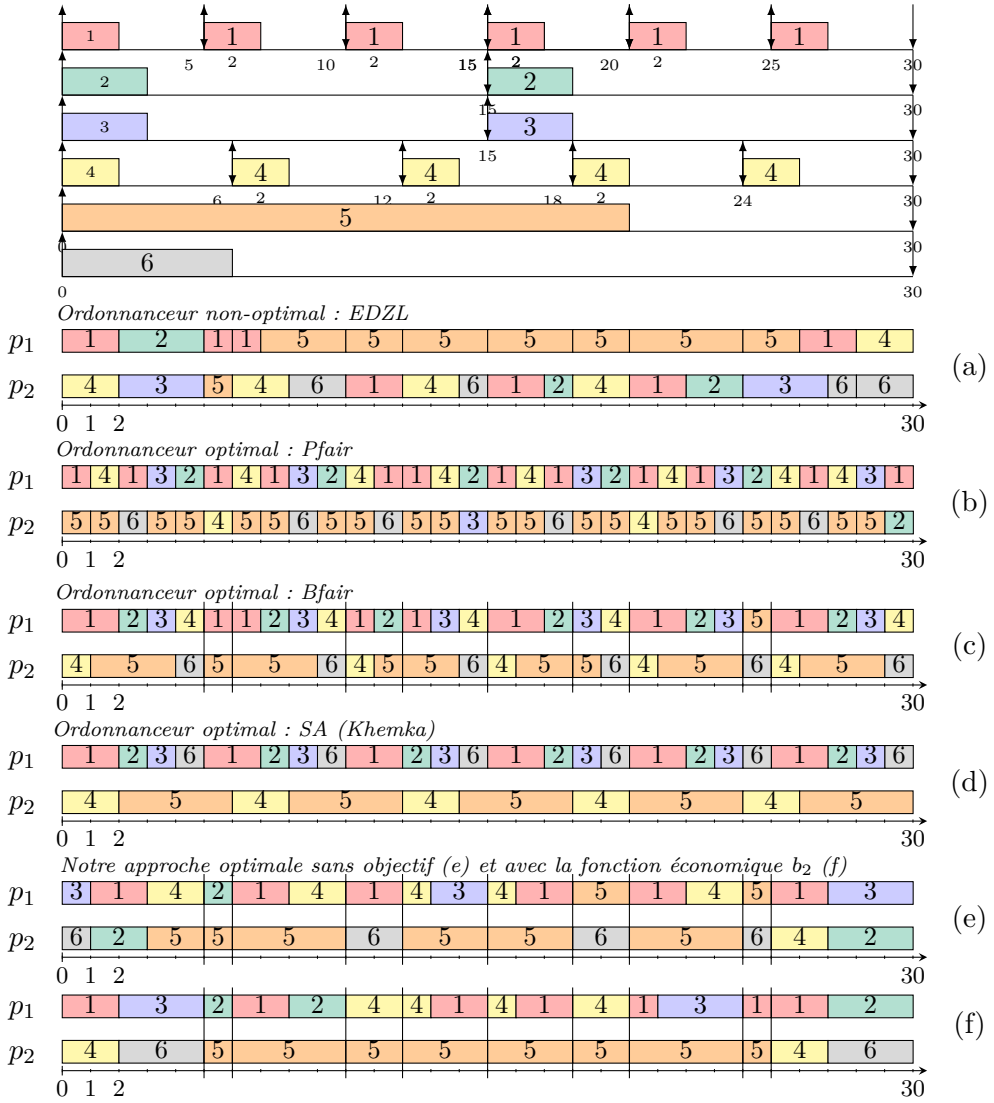


FIGURE 5.3 – Exemple d'ordonnancement avec différents algorithmes. Du haut vers le bas, le nombre de changements de contexte et de migrations (pour deux processeurs) sont respectivement de (19,5), (47,7), (40,9), (32,0), (25,6) et (19,2).

plus large de l'ensemble des tâches contrairement à tous les autres ordonnanceurs et contrairement à notre approche sans fonction économique (Fig. 5.3-e). Dans cet exemple nous produisons moins de migrations que l'algorithme EDZL. Notre solution présenté Fig. 5.3-f est fournie par la fonction économique 2. Les autres fonctions économiques sont légèrement moins puissantes mais restent intéressantes : elles produisent toutes moins de changements de contexte que les autres ordonnanceurs optimaux présentés. La table 5.5.2 montre les résultats d'ordonnancement pour chaque fonction économique. Cela confirme que :

1. Plusieurs solutions correctes peuvent être trouvées à partir de nos différentes fonctions économiques.

2. Les solutions ne sont pas nécessairement équivalentes en terme de nombre de changements de contexte de migrations produits.

Objectif	Changements de contexte	Migrations de tâches
sans (seulement Σ)	25	6
1 (minimiser b_1)	22	4
2 (minimiser b_2)	19	2
3 (minimiser b_3)	23	9
4 (minimiser b_4)	22	6

TABLE 5.3 – Comparaison d'ordonnancement entre les quatre programmes linéaires avec et sans leur fonction économique sur l'exemple de Zhu

Comme nous ordonnançons dynamiquement des ensembles de travaux intervalle après intervalle, il n'y a pas de lien direct entre eux : notre ordonnanceur dynamique ne prend en compte que l'exécution de l'intervalle en cours. Par exemple, si un sous-travail de j est exécuté à la fin d'un intervalle I_{k-1} sur le processeur p, et est également présent sur l'intervalle suivant, la décision d'ordonnancement ne prendra pas cet élément en considération pour placer le sous-travail sur le même processeur p.

Technique de permutation Afin d'atténuer ce problème et réduire ainsi les migrations de travaux, nous proposons une méthode en-ligne simple : nous utilisons le fait que, étant donné un ordonnancement valide allouant chaque tâche à un processeur, l'ordonnancement reste valide si on permute un processeur avec un autre.

Procédure
Étant donné un ordonnanceur local et J_k l'ensemble de travaux à exécuter sur I_k :

1. Au début de l'intervalle, l'ordonnanceur choisit les M premiers travaux alloués de $J \subseteq J_k$.

2. S'il y avaient des travaux ($J' \subseteq J_{k-1}$) ordonnancés à la fin de l'intervalle I_{k-1}, sélectionner les C travaux communs ($C = J \cup J'$) un par un et permuter les processeurs alloués afin d'éviter les migrations de travaux.

3. Si $C \neq \varnothing$, sélectionner les $J - C$ travaux restants et les allouer sur les processeurs restants.

Cette permutation n'est utilisée qu'une fois au début de chaque intervalle et nécessite d'être transparente pour l'ordonnanceur donc nous réalisons une allocation virtuelle des processeurs. De plus, la complexité de cette procédure est en $O(M)$. Pour l'ordonnanceur local, si nous utilisons IZL, nous n'augmentons que légèrement se complexité au début de l'intervalle. Voir par exemple la Fig. 5.4, avec $J' = \{j_{12}, j_3\}$, $J = \{j_3, j_{22}\}$ et $C = \{j_3\}$.

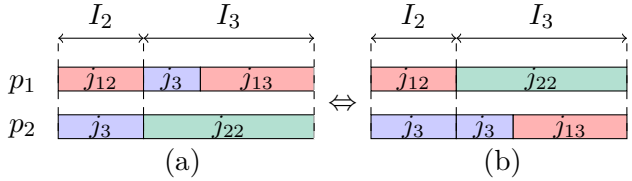


FIGURE 5.4 – (a) Fin de l'exemple d'ordonnancement de la fig. 1. (b) Ordonnancement équivalent par permutation de p_1 et p_2 au début de l'intervalle I_3.

Dans cette section, nous avons proposé un modèle amélioré en nombres entiers de notre programme linéaire. Ce modèle est composé de quatre fonctions économiques qui amènent à minimiser les préemptions de travaux sur chaque intervalle défini par les dates d'activations des travaux. En réduisant les préemptions de travaux, nous limitons également les migrations de travaux. Pour améliorer notre modèle, nous avons vu comment lier les intervalles adjacents avec une méthode simple de permutation.

5.6 Complexité du programme linéaire en nombres entiers

Nous allons montrer que notre programme linéaire est NP-difficile au sens fort. Ceci nous permettra de savoir de quel type de solveur nous aurons besoin pour résoudre les instances de ce problème, par exemple l'emploi d'un solveur sur étagère sera justifié. Appelons le *Preemptions Minimizations for Multiprocessor Schedules* (PMMS).

Soient M le nombre de processeurs, J l'ensemble des travaux, I l'ensemble des intervalles et E_J l'ensemble des intervalles où les travaux peuvent s'exécuter, PMMS nous donne une allocation par intervalle s'il en existe, telle que celle-ci minimise les préemptions des travaux entre les intervalles.

En sortie, nous obtenons l'ensemble des poids W dont tous les $w_{j,k}$ positifs correspondent à l'exécution du travail j sur l'intervalle I_k pendant $w_{j,k} * |I_k|$.

Le problème 3-partition est un problème de décision qui pour tout ensemble E de $3m$ éléments, avec $m \in \mathbb{N}$ le nombre d'intervalles de longueur $L \in \mathbb{N}$. La taille de chaque élément $s : E \to \mathbb{N}$, est telle que $\frac{L}{4} < s(e) < \frac{L}{2}, \forall e \in E$ et l'on sait que leur somme est telle que $\sum_{e \in E} s(e) = mL$. Existe-t-il une partition de E en m sous-ensembles disjoints $\{E_1, E_2, ..., E_m\}$ tels que :

$$\forall i \in \{1, ..., m\}, \sum_{e \in E_i} s(e) = L$$

On peut en déduire que nécessairement chaque intervalle contient exactement 3 éléments : $\forall i, |E_i| = 3$.
Ce problème est connu pour être NP-complet au sens fort [GJ79].

Proposition 1. *Le problème PMMS est NP-difficile au sens fort, même pour 1 processeur.*

Démonstration. Nous procédons par restriction au problème 3-partition.
On considère un système composé d'un processeur $M = 1$, d'un ensemble de tâches Γ tel que $\Gamma = \{T_v\} \cup \{\Gamma_{3m}\}$. T_v est une tâche périodique de temps d'exécution quelconque $T_v.e$ et de période $T_v.p = L + T_v.e$. L'ensemble Γ_{3m} est composé de $3m$ tâches de période commune, $\forall T \in \Gamma_{3m}, T.p = m * (L + T_v.e)$ et de temps d'exécution $\forall T \in \Gamma_{3m}, \frac{L}{4} < T.e < \frac{L}{2}$. De plus, on sait que $\sum_{T \in \Gamma_{3m}} T.e = mL$.

La tâche T_v est composée de m travaux $\{j_v^1, ..., j_v^m\}$ alors que les tâches de Γ_{3m} sont constituées chacune d'un seul travail, nous les notons $\{j^1, ..., j^{3m}\}$. On cherche à montrer que l'instance décrite est une oui-instance *si et seulement si* on peut placer sans division l'ensemble des travaux des tâches de Γ_{3m} sur un ensemble disjoints (consécutifs) d'intervalles $I_1, ..., I_m$, tel que $\forall k \in \{1, ..., m\}$ avec J_k l'ensemble des travaux de l'intervalle I_k :

$$\sum_{j \in J_k} w_{j,k} * |I_k| = L + j_v^k.e \tag{5.15}$$

Bien que $\forall k, |I_k| = L + T_v.e$, un travail de T_v occupe déjà $T_v.e$ unités de temps à chaque intervalle. On peut donc réécrire l'Eq. 5.15, $\forall k \in \{1, ..., m\}$:

$$\sum_{j \in J_k \setminus \{j_v^k\}} w_{j,k} * |I_k| = L \tag{5.16}$$

Rappelons que l'équation 5.13 de notre problème PMMS permet de s'assurer qu'un travail est bien exécuté sur un intervalle, elle s'écrit :

$$\forall j, \forall k, w_{j,k} * |I_k| \geq min(\alpha * j.e, |I_k|) * x_{j,k} \tag{5.17}$$

Comme nous savons que $\forall j, j.e < L$ car $\forall j, j.e < \frac{L}{2}$, alors $min(\alpha * j.e, |I_k|) = \alpha * j.e$ dans notre cas. Nous pouvons donc la réécrire :

$$\forall j, \forall k, w_{j,k} * |I_k| \geq \alpha * j.e * x_{j,k} \tag{5.18}$$

Ceci traduit qu'il existe des intervalles pour lequel chacun des travaux sera exécuté. Cependant, nous cherchons à ce que chaque travail soit présent sur un intervalle. Le paramètre α mesure le degré de division des intervalles : en effet, plus α est proche de 1, moins les travaux seront divisés. Cela revient donc à poser $\alpha = 1$. Si $\alpha = 1$ alors :

$$\forall j, \exists k, w_{j,k} * |I_k| = j.e \tag{5.19}$$

Supposons qu'une 3-partition existe (Eqs. 5.16 et 5.19), alors tous les travaux sont placés dans leur intégralité dans l'un des intervalles et peuvent donc être exécutés sans préemption sur chacun des intervalles de longueur L. Notre problème PMMS, qui revient à un problème de satisfaction de contrainte, possède donc une solution.

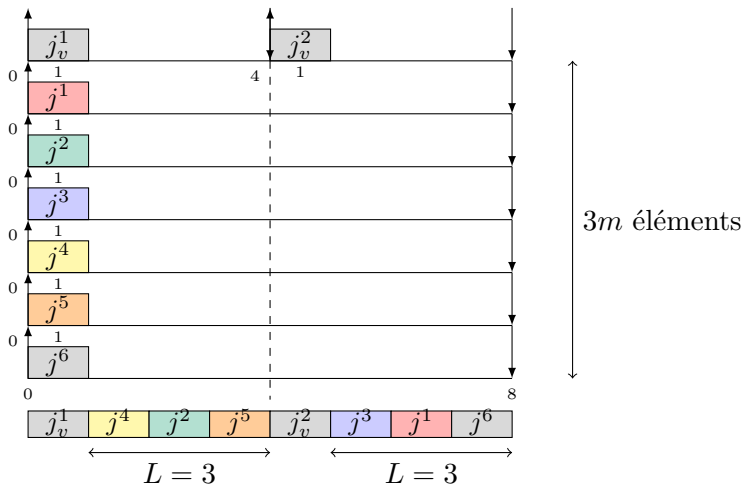


FIGURE 5.5 – Exemple de restriction du PMMS avec $M = 1$, $\alpha = 1$ au problème 3-partition pour $L = 3$, $m = 2$.

Supposons maintenant qu'une 3-partition n'existe pas. Cela revient à dire qu'il existe m' le plus grand entier tel qu'il existe un ensemble disjoints d'intervalles qui satisfont les Eqs. 5.16 et 5.19, $\forall k \in \{1, ...m'\}$.
Il faut nécessairement que $m' < m - 1$. Sinon pour $m' < m$ tel que $m' = m - 1$, on vérifierait une 3-partition : on aurait $m - 1$ intervalles allouant les travaux sans préemption, et comme il ne

resterait que L unités de temps à allouer, l'intervalle I_m serait forcément alloué avec les travaux restants non préemptés.

Donc pour $m' < m-1$, on ne trouve plus d'allocation sur les intervalles restants permettant d'allouer L unités de temps sans qu'un travail ne doive être divisé sur plusieurs intervalles. Or $\alpha = 1$, notre problème PMMS, qui revient à un problème de satisfaction de contrainte, ne possède donc pas de solution.

Le problème de 3-partition peut être résolu par un algorithme capable de résoudre le problème PMMS. La NP-complétude de ce dernier suit par *restriction* le problème de 3-partition, lui-même NP-complet au sens fort.

Par conséquent, il n'existe aucun algorithme polynomial ou pseudo-polynomial pour le problème PMMS, à moins que $P = NP$. □

5.7 Expérimentation

Dans l'objectif de mesurer la pertinence pratique de notre approche en terme de préemptions et de migrations de tâches, nous avons conduit une étude expérimentale basée sur la simulation. À cette fin, nous avons développé un simulateur d'ordonnanceurs sur l'hyperpériode.

5.7.1 Contexte et configuration de la simulation

Nous considèrons ici quatre ordonnanceurs optimaux : Pfair, Bfair, SA (celui de Khemka) et notre approche. Dans notre simulation, nous avons utilisé des jeux de 100 tâches pour chaque pas d'utilisation système. Chaque tâche est générée uniformément de manière aléatoire pour les périodes et les temps d'exécution. Les périodes sont comprises entre 10 et 100, et les temps d'exécution entre 1 et la période. Nous avons choisi des valeurs entières pour les périodes et les temps d'exécution, nécessaires pour Pfair et Bfair. L'utilisation de chaque tâche (i.e. $u_i = T_i.e/T_i.p$) est dans une fourchette $[0.01, 1]$ et l'utilisation système (i.e. $U = \sum_i u_i/M$) varie entre 0.1 et 1. Nous rejetons tout ensemble de tâches produisant une hyperpériode supérieure à 2^{32}.

Nous considérons les règles de comptage suivantes :

- On compte chaque changement de contexte [4] à l'exception des sous-travaux exécutés sur le même processeur sur des intervalles adjacents (présents à la fin de l'un et au début du suivant), voir pour exemple j_3 sur la Fig. 5.4-b.
- On compte une migration de travail si un processeur p exécute un travail j au temps t et une autre processeur p' exécute le même travail au temps t' ($t' > t$). Nous comptons également une migration de travail lorsque le travail n'appartient pas à la même activation de la tâche.

Afin de mesurer le nombre moyen de changements de contexte et de migrations de travaux, 100 simulations sont effectuées (avec des jeux de tâches différents).

Nous générons alors les contraintes correspondantes pour chaque ensemble de tâches pour les

4. On peut noter que l'on propose de compter le nombre de changements de contexte au lieu du nombre de "préemptions de travaux" pour faciliter les mesures. Cela n'apporte que peu de différence car dans notre cas, minimiser les préemptions de travaux implique moins de changement de contextes. Dans la fin de ce chapitre, nous utiliserons les deux termes sans distinction.

quatre fonctions économiques. Ces contraintes représentent notre programme linéaire en nombres entiers et réels. Comme nous cherchons la valeur critique de α, nous utilisons une recherche par dichotomie. Le solveur fournit en réalité soit aucun résultat (programme infaisable à cause d'une valeur trop haute de α) ou les poids des sous-travaux pour chaque intervalle. Lorsqu'une solution est trouvée, nous ordonnançons tous les poids des travaux sur chaque intervalle avec IZL. Nous choisissons finalement pour chaque U, la fonction économique qui produit en moyenne le meilleur ordonnancement en termes de nombre de changements de contexte et de migrations de tâches engendrés. Notons qu'il est tout à fait possible de ne pas trouver de solution pour un ensemble particulier de tâches dans le temps imparti (en effet, nous limitons le temps de résolution du solveur à 60 sec pour chaque jeu de tâches). Le solveur utilisé est ILOG Cplex (IBM). Trois cas de résolution peuvent apparaître pour chaque fonctions économiques :

- Le solveur trouve la meilleure solution entière (le meilleur cas),
- Après le temps imparti, le solveur trouve une solution approchée (près de l'optimal).
- Après le temps imparti, le solveur ne trouve pas de solution.

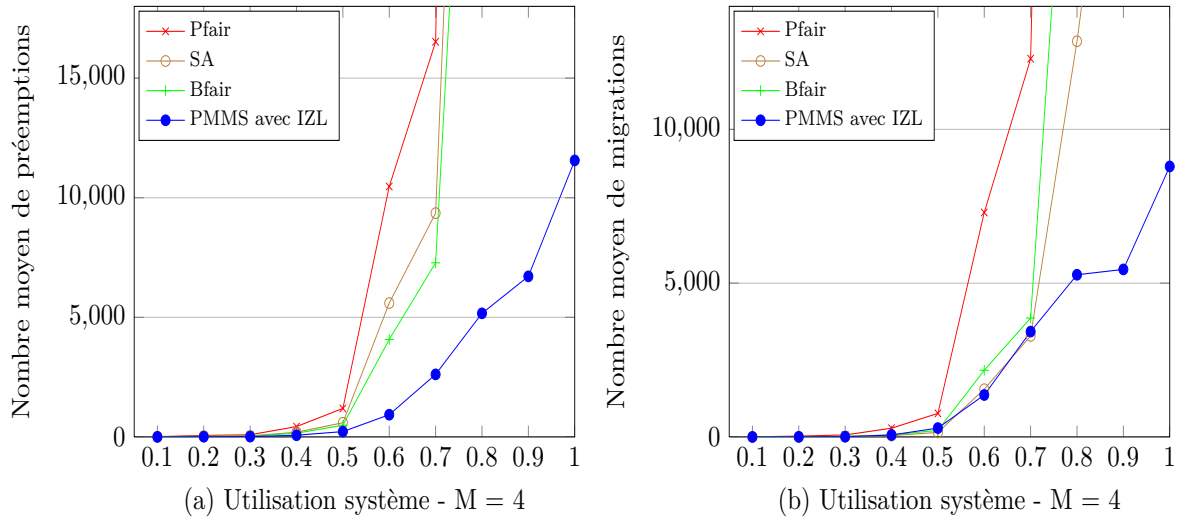
Comme nous le verrons dans la section suivante, les deux derniers cas se produisent de manière marginale dans nos simulations pratiques.

5.7.2 Résultats

Nous appelons notre approche : PMMS avec IZL. Les figures 5.6-a et 5.6-b montrent que les résultats des préemptions/migrations de travaux pour chaque algorithme en fonction de l'utilisation système : l'axe horizontal correspond donc au taux d'utilisation système et l'axe vertical correspond au nombre moyen de préemptions de travaux *sur l'hyperpériode* normalisé par le nombre d'ensemble de tâches. Les ordonnancements de Pfair produisent un nombre de préemptions à partir de $U = 0.6$, donc nous avons décidé de tronquer les plus hautes valeurs sur les courbes. On peut noter le même comportement pour ceux de Bfair et de SA. En Fig. 5.6-a, notre approche est celle avec la plus basse suivie de Bfair, SA et Pfair. Les préemptions de travaux sont entre deux et trois fois inférieures aux autres ordonnanceurs optimaux pour une utilisation système compris entre $U = 0.1$ et 0.5. Pour de plus hautes valeurs, cette proportion augmente.

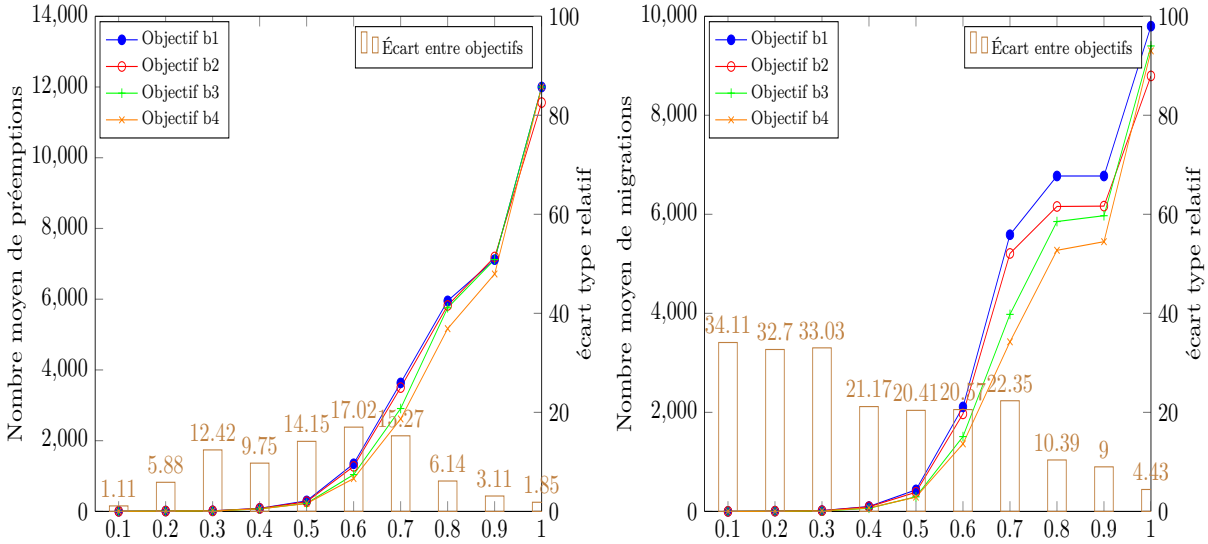
La figure 5.6-b montre les résultats des migrations de tâches pour chaque algorithme. Pour $U = 0.1$: Bfair arrive premier suivi de notre approche puis des autres. À partir de $U = 0.2$ et jusqu'à 0.7, notre approche et celle de Khemka sont très proches, suivi par les autres ordonnanceurs. Pour des valeurs plus hautes, notre approche domine les autres. Globalement par rapport à Pfair, les algorithmes Bfair et SA produisent deux fois moins de préemptions/migrations, alors que notre approche produit deux fois moins de préemptions/migrations pour des taux d'utilisation faibles et jusqu'à un facteur 25 pour des taux d'utilisation élevés.
Nos quatre approches génèrent des ordonnancements différents qui ne décrivent pas la même tendance en fonction du taux d'utilisation système : par exemple les meilleurs résultats sont obtenus soit par l'objectif 3 (minimisation des présences des travaux) pour un faible taux d'utilisation ou soit par l'objectif 4 (minimisation des présences et préemptions) pour des taux d'utilisation élevés. La Fig. 5.6 présente uniquement l'allure de la fonction économique qui pour chaque pas d'utilisation présentait les meilleurs résultats. Avec la Fig. 5.7, nous présentons l'allure des résultats pour les quatre fonctions économiques.

L'écart type relatif (écart type sur moyenne) est présenté sur la Fig. 5.7 en complément de l'évolution du nombre de préemptions/migrations en fonction du taux d'utilisation. Cet écart



(a) Utilisation système - M = 4 (b) Utilisation système - M = 4

FIGURE 5.6 – Nombre moyen de préemptions (a) et de migrations (b) de travaux entre notre approche et 3 autres algorithmes optimaux pour $M = 4$, $U = [0.1, 1]$.



(a) Utilisation système - M = 4 (b) Utilisation système - M = 4

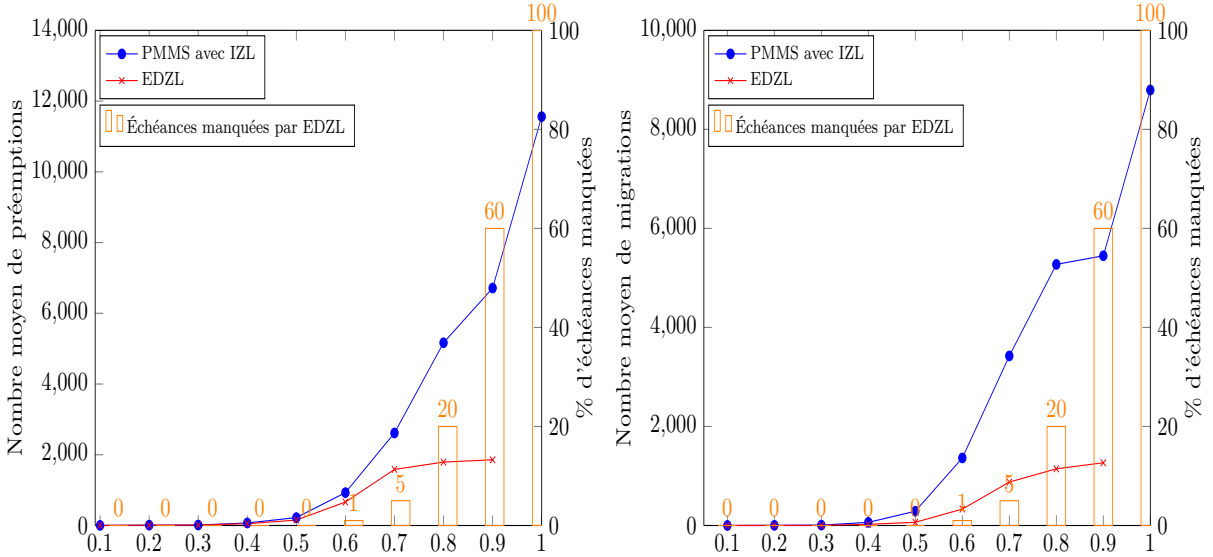
FIGURE 5.7 – Évolution des résultats selon les fonctions économiques $M = 4$, $U = [0.1, 1]$: (a) nombre moyen de préemptions ; (b) nombre moyen de migrations.

type relatif est calculé entre les quatre fonctions économiques présentées à la section précédente. Elle atteint au maximum 17% pour le nombre moyen de préemptions engendrés et de 34% pour le nombre moyen de migrations engendrés. Ceci traduit le fait que choix de la fonction économique a un impact sur les résultats mais celui-ci reste relativement limité dans le cadre des préemptions. Notons également que l'écart relatif est légèrement inexact pour des fortes valeurs d'utilisation car comme nous allons le détailler, le nombre de jeux de tests est légèrement inférieur pour les hautes valeur d'utilisation système.

Le facteur α varie en pratique entre 0 et 1 et comme attendu, plus le taux d'utilisation augmente plus les valeurs d'α diminuent également : des valeurs proches de zéro sont souvent trouvées pour

des taux d'utilisation système élevés.

Nous nous intéressons également à savoir à quelle "distance" se trouve notre approche des bornes inférieures du nombre de préemptions et de migrations. Sans les connaître, nous pouvons tout de même nous comparer à EDZL, un ordonnanceur non-optimal (concernant *les tâches périodiques*) connu pour être plus efficace que Global-EDF et produisant peu de préemptions de tâches [WCL+07].



(a) Utilisation système - M = 4 (b) Utilisation système - M = 4

FIGURE 5.8 – Nombre moyen de préemptions (a) et de migrations (b) entre notre approche et EDZL pour $M = 4$, $U = [0.1, 1]$. En deuxième axe des ordonnées figurent le pourcentage de jeux de tâches surlequels EDZL a raté des échéances.

Les simulations montrent des résultats proches pour de faibles taux d'utilisation, voir Fig. 5.8. Nous générons tout de même plus de préemptions que EDZL : entre 3% et 31% pour un taux d'utilisation entre 0.1 et 0.7. Pour des taux plus élevés, le ratio augmente jusqu'à 300% mais dans ces cas là, EDZL commence à manquer des échéances : le nombre d'ordonnancements corrects chutent significativement alors que notre approche garantit la ponctualité. À $U = 0.8$, EDZL échoue sur 20% des jeux de tâches et jusqu'à près de 100% pour $U = 1$. Concernant les migrations, les ordonnancements de EDZL produisent les chiffres les plus faibles excepté pour une utilisation système de $U = 0.1$. Pour des taux plus élevés, notre approche produit jusqu'à cinq fois plus de migrations que EDZL, mais EDZL manque ses échéances.

Comme prévu, certains jeux de tests très contraints fournis au solveur ne permettent pas de trouver de solution pour nos quatre fonctions économiques pendant la phase de résolution, mais le nombre de cas est relativement faible et ne se produit que pour des taux d'utilisation système élevés, comme montré dans la table 5.4.

L'autre cas rencontré est celui des solutions approchées. La limite de temps investi pour laisser Cplex résoudre est relativement court (60 secs). Il est raisonnable d'augmenter le temps de résolution si l'utilisateur veut disposer toujours de solutions ou uniquement de solutions exactes (optimales). Il y a un compromis à trouver entre le temps investi et la recherche de solutions optimales.

Utilisation système	Pourcentage de solutions manqués (avec 60 secs pour le solveur)
0-0.5	0%
0.6	3%
0.7	6%
0.8-0.9	8%
1	10%

TABLE 5.4 – Nombre de jeux de tâches non résolus fonction du taux d'utilisation système

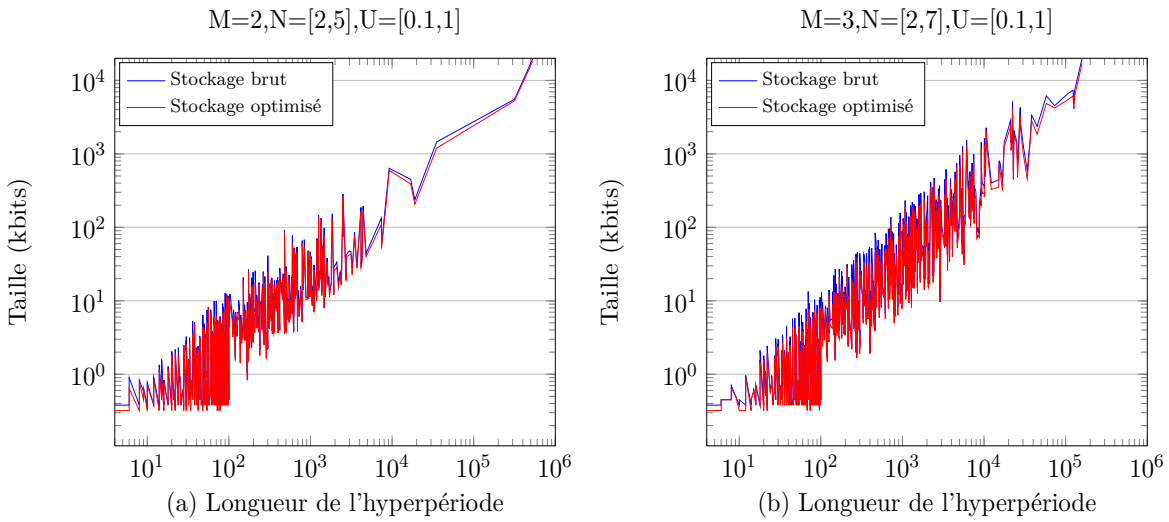
Pendant nos expérimentations, la partie hors-ligne (solveur, recherche dichotomique) a pris entre 10 secs et 10 mins par objectif : ce temps augmente comparativement au taux d'utilisation système. Nous proposons donc un comparatif du temps sur la partie hors-ligne fonction du taux d'utilisation système en table 5.5.

Utilisation système	Durée de la partie hors-ligne
0-0.5	10-20 s
0.6	1-2 min
0.7	1-5 min
0.8-1	1-10 min

TABLE 5.5 – Durée effective de la partie hors-ligne (solveur, recherche dichotomique) pour chaque objectif fonction du taux d'utilisation système

Nous allons nous intéresser à l'empreinte mémoire nécessaire pour appliquer notre approche : celle-ci nécessite, pour l'exécution en-ligne, de stocker les poids $w_{j,k}$ et la longueur des intervalles $|I_k|$. La méthode brute est de stocker tous les poids strictement positifs et la durée des intervalles sur 64 bits. Une simple optimisation est de ne stocker que les poids positifs différents et de les différencier par le numéro l'intervalle sur lequel ils sont. Nous avons conduit une simulation de manière similaire à la précédente. Nous avons décider d'évaluer l'empreinte mémoire suivant les paramètres suivants : nous considérons un modèle de tâche périodique à échéance implicite, avec des périodes comprises entre 2 et 100, de temps d'exécution entre 1 et la période. Le nombre de tâches varie entre 1 et 10 et le nombre de processeurs entre 1 et 3. Enfin, nous limitons l'hyperpériode à 500000 et le taux d'utilisation système est compris entre 10 et 100 %.
La figure 5.9 présente les résultats avec une cible à deux et trois processeurs. Les échelles sont logarithmiques afin de pouvoir distinguer l'évolution plus facilement.

En bleu est tracée l'évolution du stockage mémoire nécessaire pour le stockage brut, et en rouge pour la méthode légèrement optimisée. Ces courbes se superposent en grande partie, ce qui veut dire que la méthode d'optimisation proposée permet de gagner légèrement en mémoire, mais globalement, nous constatons que la majorité des $w_{j,k}$ ont des valeurs différentes, et qu'il est difficile de réduire la taille en les factorisant. Nous observons tout de même une diminution moyenne de 7% avec un extremum à 44% pour $M = 2$ et une diminution moyenne de 13% avec un extremum à 52% pour $M = 3$.
Néanmoins, on constate d'une manière générale que l'empreinte mémoire varie assez peu sauf pour de très grandes valeurs de l'hyperpériode, qui n'ont qu'un intérêt théorique. L'empreinte est majoritairement comprise entre 1 et la dizaine de kbits pour $M = 2$ et 1 et 100 kbits pour $M = 3$, ce qui est une valeur envisageable pour une cible embarquée, par exemple dans un contexte industriel temps-réel strict [JDLP10].



(a) Longueur de l'hyperpériode (b) Longueur de l'hyperpériode

FIGURE 5.9 – Évolution de l'empreinte mémoire en fonction de la longueur de l'hyperpériode pour une méthode de stockage brute et légèrement optimisée : a) pour $M = 2$, b) pour $M = 3$.

5.8 Conclusion

Nous avons proposé une méthode d'ordonnancement disposant d'une meilleure extensibilité en minimisant les surcoûts : minimisant les changements de contexte et en réduisant la complexité en-ligne. Contrairement aux approches antérieures, notre méthode est basée sur une construction hybride (hors-ligne et en-ligne) : l'une hors-ligne pour placer les travaux sur des intervalles successifs, l'autre en-ligne pour les exécuter dynamiquement à l'intérieur de chaque intervalle. Chaque intervalle étant défini par les différentes échéances des travaux. Notre problème d'ordonnancement est équivalent à l'ordonnancement d'un ensemble fini de travaux sur des intervalles consécutifs et il est donc possible de se baser sur la représentation par poids et par intervalle [LDAVN08]. Cette représentation permet d'exprimer les contraintes temps-réel sous forme d'un programme linéaire. La résolution de ce programme fournit un test exact de faisabilité et apporte un ensemble de poids valides pour chaque intervalle.

Nous avons proposé un modèle linéaire en nombres entiers (PMMS) afin d'exprimer la présence et la préemption d'un travail entre plusieurs intervalles. Pour compléter le modèle, nous avons étudié quatre fonctions économiques qui mènent à des résultats d'ordonnancement corrects mais différents en termes d'optimisation. La partie en-ligne consiste à exécuter des travaux avec un ordonnanceur approprié. Nous en avons présenté un de faible complexité (IZL), dynamique et générant peu de préemptions et de migrations de tâches.

Notre approche permet de réduire significativement le nombre de préemptions et migrations de travaux pour le modèle de tâches périodiques. Les résultats montrent que nous générons moins de préemptions et migrations que d'autres ordonnanceurs optimaux. Cependant, il doit être souligné que notre approche peut nécessiter un investissement significatif (plusieurs minutes) avec l'augmentation de l'hyperpériode, du nombre de tâches ou de l'utilisation système : ceci reste acceptable car ne concerne que la partie hors-ligne. Cependant, les améliorations induites en-ligne valent cet investissement. De plus, des durées relativement longues pour la compilation sont courantes dans le domaine des systèmes embarqués en fin de cycle de développement. Enfin, il est toujours possible de trouver un compromis correct entre le temps investi hors-ligne et le temps libre de l'utilisateur.

Nous avons examiné la complexité de calcul de notre programme linéaire, celui-ci est NP-difficile au sens fort. D'un point de vue pratique, nous avons étudié également l'empreinte mémoire des données de travail de l'ordonnanceur (les ensembles de poids) qui s'avère être de l'ordre de quelques kbits pour des instances réalistes.

Pour la suite des travaux, nous voulons compléter nos expérimentations en faisant varier davantage le nombre de processeurs et en ajoutant d'autres ordonnanceurs optimaux tels que [FKY08] ou non non-optimaux.

Il serait intéressant d'étendre le test d'ordonnançabilité pour prendre en compte les coûts de préemptions et de migrations. Nous avons accès, de par la partie hors-ligne de notre approche, à une borne inférieure du nombre de préemptions. En effet, la borne inférieure est donnée pour chaque travail de chaque tâche par la variable y_j. La borne supérieure des préemptions est la même que celle de Bfair ou DP-Wrap.

Nous voulons appliquer notre méthode à des modèles de tâches plus généraux comme le modèle de tâches time-triggered [LDAV10]. Même si les tâches disposent de temps d'exécution et d'échéances différentes selon les travaux exécutés, notre approche reste applicable dès lors que l'on peut connaître statiquement tous les comportements temporels possibles.

Enfin, une implémentation de cette approche sur un outil prototype permettrait de mesurer les performances réelles par rapport aux autres algorithmes globaux à priorité fixe ou dynamique.

Chapitre 6

Conclusion et perspectives

Sommaire

Afin de conclure cette thèse, nous allons tout d'abord résumer les problématiques traitées que nous venons de présenter, les contributions que nous apportons et l'évaluation des résultats obtenus. Nous apportons des solutions suivant deux axes concernant les systèmes embarqués temps-réel critiques, le premier concerne la flexibilité dans l'allocation des tâches suite aux défaillances et le second sur l'optimisation des performances locales. Nous présentons ensuite les perspectives envisageables à court et à moyen termes sur ces sujets.

6.1 Conclusion générale

Nous avons proposé une méthode établissant une politique hors-ligne pour la réallocation en-ligne des tâches et de leurs répliques. Cette méthode tend à améliorer la tolérance aux fautes permanentes dans l'exécution de systèmes temps-réel distribués. À chaque défaillance, l'objectif est de reconfigurer le système de manière à atteindre une allocation dite de référence optimisée pour les ressources restantes. Pour de tels systèmes, nous devons nous assurer que ces *reconfigurations* sont sûres, que les mécanismes utilisés pour y parvenir sont déterministes et efficaces.

Nous avons identifié les mécanismes nécessaires appropriés (réplication, swact, migration). L'allocation des tâches et répliques doit être dimensionnée pour prendre en compte les ressources nécessaires pour l'exécution de ces reconfigurations. Une formulation linéaire permet de spécifier les allocations en présence de ces mécanismes dans un cadre général (contraintes de ressources simples). La migration est l'opération la plus longue à effectuer, nous avons donc cherché à les minimiser, notamment via l'utilisation d'une technique en temps polynomial permettant de trouver le nombre de migrations minimum à effectuer pour atteindre une allocation de référence. Afin d'être exhaustif sur la gestion des scénarios de défaillances, nous montrons comment explorer tous les scénarios possibles par niveau de défaillances, cette exploration étant facilitée lorsque l'architecture dispose de ressources identiques sur chacun des nœuds.

Ces reconfigurations doivent être bornées dans le temps afin de garantir une maîtrise du temps reconfiguration, utile par exemple pour l'évaluation de la fiabilité globale du système. À notre connaissance, aucun mécanisme de migration de tâche n'a encore été proposé lorsque le système considéré est soumis à des contraintes temps-réel strictes.

Nous avons tout d'abord montré qu'il est possible d'opérer des migrations dans ce contexte, de manière sûre i.e. en respectant toutes les échéances des tâches en exécution, en temps borné et sans temps de gel. L'approche que nous avons développée repond à cette problématique. Le temps de gel est évité par la connaissance des périodes d'inactivité de la tâche migrante, tandis qu'une garantie sur le temps de transfert est possible dans notre contexte, où nous disposons de la description statique des contraintes temporelles de l'ensemble des tâches. Pour assurer la ponctualité des tâches en présence de migrations, nous avons développé une approche basée sur l'utilisation d'une tâche système en charge d'effectuer la migration des tâches. Nous proposons différentes techniques de transfert pour y parvenir. Nous montrons comment assurer la faisabilité de la migration en fonction des caractéristiques de la tâche migrante et de celles du réseau. Afin de garantir l'exécution correcte des tâches du système en présence de migration, nous exprimons les caractéristiques temporelles de cette tâche système. Ceci nous permet alors de considérer cette migration comme une tâche temps-réel additionnelle et ainsi de pouvoir vérifier l'ordonnançabilité globale. Afin de vérifier la pertinence de cette approche, nous avons étudié l'impact de ces migrations sur l'exécution, basé sur une métrique évaluant le nombre de préemptions du système avec et sans migration. Notre évaluation est réalisée par simulation et les résultats montrent que pour des tâches migrées de tailles mémoire raisonnables pour un contexte embarqué, le surcoût reste relativement faible par rapport au nombre de migrations réalisé.

Nous nous sommes intéressés également à l'efficacité de l'exécution des tâches au niveau local d'un nœud disposant d'un multicœur. Nous proposons des ordonnancements optimaux multiprocesseur minimisant le nombre de changements de contexte engendrés et avec une complexité en-ligne faible. Notre méthode est basée sur une construction hybride (hors-ligne et en-ligne) : une phase hors-ligne pour placer les travaux sur des intervalles successifs, l'autre phase en-ligne pour les exécuter dynamiquement à l'intérieur de chaque intervalle.

Nous proposons un modèle linéaire en nombres entiers afin d'exprimer la présence et la préemption d'un travail entre plusieurs intervalles. Pour compléter le modèle, nous proposons quatre fonctions économiques qui mènent à des résultats d'ordonnancement corrects mais différents dans l'expression de l'optimisation. La partie en-ligne consiste à exécuter des travaux avec un ordonnanceur approprié. Nous proposons un algorithme, IZL, pour sa faible complexité, son caractère dynamique ainsi que sa capacité à générer peu de préemptions et de migrations locales de tâches. Les résultats obtenus montrent qu'il est possible de réduire significativement le nombre de changements de contexte par rapport à d'autres approches existantes. Nous soulignons que cette approche peut nécessiter un investissement en temps significatif dans sa partie hors-ligne mais acceptable. Il est toujours possible de trouver un compromis correct entre le temps investi hors-ligne et le temps dont dispose l'utilisateur. La complexité de calcul du programme linéaire est NP-difficile au sens fort, justifiant l'utilisation d'un solveur sur étagère (CPLEX). Enfin, l'empreinte mémoire nécessaire à la partie en-ligne (ensemble de poids) a été évaluée à quelques kbits pour des instances réalistes.

À travers cette thèse, nous proposons des réponses pour résoudre certains problèmes relatifs à l'étude et la conception de systèmes embarqués temps-réel distribués. Notre objectif est de les rendre plus tolérants aux fautes et plus performants. Comme déjà souligné, ce type de système est complexe à concevoir et à maîtriser. Nous espérons avoir apporter des approches intéressantes pour la reconfiguration du système dans un objectif de tolérance aux fautes, son application à

travers les migrations temps-réel et pour l'efficacité de l'exécution des tâches dans les systèmes multiprocesseurs.

6.2 Perspectives

6.2.1 Flexibilité dans l'allocation des tâches

A propos de notre approche de reconfiguration, celle-ci peut être encore utile à la tolérance aux fautes dans le cadre de la réintégration d'un nœud initialement considéré comme fautif et à nouveau fonctionnel après un redémarrage. On retrouve ce type d'approche dans les algorithmes de *membership* [SBS+11] dont le rôle est d'exclure ou d'intégrer les nœuds au système. Par rapport à notre travail, il faudrait envisager les migrations de tâches nécessaires non plus entre une allocation suite à défaillance et une allocation de référence mais entre deux allocations de référence de niveau différent (de nombre de nœuds différents).

Nous présentons maintenant plusieurs autres contextes applicatifs où cette méthode peut être utile : la gestion de l'énergie ou des modes de fonctionnement, les télécommunications par la présence de contraintes de type sac à dos ou dans les multicœurs clusterisés avec des processus de type "dataflow".

Flexibilité pour la gestion de l'énergie ou des modes La flexibilité opérationnelle peut se retrouver dans un contexte embarqué pour la gestion de la consommation d'énergie : une politique d'économie d'énergie pourrait vouloir désactiver temporairement les ressources, un nœud par exemple. Il faudrait alors éventuellement redistribuer la charge sur les nœuds actifs restants. On peut également envisager des reconfigurations pour les changements de mode, non pas à l'échelle d'un processeur, mais à celle d'un système distribué. Les changements de modes sont utiles pour l'exécution de différentes phases d'un système. Ceci se traduit sur le plan temps-réel par des ensembles de tâches distincts ou avec des temps d'exécution différents. Une réallocation des tâches sur le système distribué est alors éventuellement nécessaire pour l'équilibre de la charge par exemple.

Flexibilité pour les Télécommunications (contraintes de type sac à dos) La méthode présentée s'applique très bien pour les systèmes commutés telecom à haute disponibilité [Sir03]. La technique de permutation de nœuds présentée en Section 3.3.3 a été utilisée dans ce contexte et permet typiquement de réduire de l'ordre de 20% le nombre de migrations trouvé par rapport à un algorithme qui ne l'utiliserait pas [SPG03]. Un état admissible correspond à l'allocation de tâches et répliques de traitement d'appels tel que les contraintes de ressources de TMU (*Traffic Management Units*) sont respectées et tel qu'il n'y aient pas de réplique passive sans tâche active.

La charge de traitement d'appels est seulement protégée contre les défaillances de TMU qui sont suffisamment espacé dans le temps. Les capacités de redondance requises afin d'assurer le basculement des répliques passives ne dépend pas linéairement de la consommation totale des tâches actives. Une consommation *effective* est associée à chaque réplique passive, elle représente une fraction potentiellement égale à zéro de la consommation de la tâche active, qui correspond au reste. Soit A l'ensemble des tâches actives et P l'ensemble des répliques passives, la consommation en ressource d'une TMU t est donnée par :

$$\sum_{a \in A : f^{(A)}(a) = t} w_a + \sum_{p \in P : f^{(P)}(p) = t} w_p + \max_{t' \neq t} \sum_{\substack{p \in P : f^{(P)}(p) = t \\ \wedge f^{(A)}(\alpha(p)) = t'}} w'_p, \tag{6.1}$$

où w_a, w_p et w'_p représentent respectivement la consommation active de a et les consommations effectives et redondantes de la réplique passive p dans la ressource, où $f^{(A)} : A \longrightarrow T$ et $f^{(P)} : P \longrightarrow T$ représentent respectivement l'allocation de la tâche active et de la réplique sur l'ensemble de TMU T, et, où $\alpha : P \longrightarrow A$ est tel que $\alpha(p)$ indique la tâche active de la réplique p. Notons que $w_{\alpha(p)} = w_p + w'_p$.

Dans ce contexte, les problèmes de l'évaluation d'allocations de qualité de même que l'évaluation de l'impact de l'ordonnancement des migrations ont été largement adressés (voir respectivement [SPG03] et [SCKN07]). Les algorithmes présentés dans ces études peuvent directement être utilisés comme oracles dans la méthode décrite ici qui présente donc une alternative intéressante aux mécanismes dynamiques de tolérance aux fautes qui étaient implémentés dans le système mentionné ci-dessus.

Flexibilité pour les architectures multi/manycœurs La méthode présentée apparaît également être très pertinente dans l'instanciation de réseau de processus de type "dataflow" (un réseau de processus communicants via des canaux FIFO [LP95]) sur une architecture multicœurs. En particulier, cette méthode peut être utile dans la conception et le développement d'architectures clusterisées massivement multicœurs (avec quelques dizaines de clusters comprenant quelques centaines de cœurs). Dans ce contexte, trouver l'allocation appropriée du réseau de processus sur une architecture de type cluster nécessite la résolution de plusieurs problèmes combinatoires (partitionnement de graphes de capacité des nœuds, allocation quadratique avec des contraintes multi-flot, voir [SD08]). De nouveau, la méthode présentée peut servir de base pour la conception de framework de tolérance aux fautes niveau cluster, en utilisant des algorithmes existants (ou conçus indépendamment) comme oracles.

6.2.2 Mécanismes de migration

A propos des mécanismes de migration de tâches présentés, il reste à évaluer leur efficacité sur un autre critère basé sur une métrique réseau pour identifier les politiques permettant le meilleur équilibre de la charge réseau. Il serait intéressant d'étudier d'autres stratégies de migrations pour des modèles de tâches périodiques communicants [TC94], avec dépendances entre les travaux et dépendances de données. L'utilisation de tels modèles va demander de revoir les politiques de migrations qui transfèrent l'espace mémoire travail par travail en prenant compte par exemple des zones de mémoire partagée pour les communications.

Enfin, l'étude de la faisabilité des migrations pourrait être étendue pour tenir compte des effets de cache. Nous prévoyons de caractériser l'impact des effets de cache dûs à la migration dans l'analyse de faisabilité. Ainsi le temps de transfert ne serait plus un simple rapport taille mémoire sur débit mais prendrait aussi un délai pour le chargement des données en cache sur le nœud de destination.

6.2.3 Ordonnancement multicœurs

Concernant notre approche pour l'ordonnancement multiprocesseur, il serait intéressant d'évaluer l'influence du nombre de processeurs et d'inclure d'autres algorithmes optimaux tels que [FKY08].

Le programme linéaire présenté, PMMS, s'applique actuellement à une politique globale d'ordonnancement et pour processeurs identiques. L'une des extensions pourraient être l'utilisation de processeurs dits homogènes dont la fréquence d'horloge de chacun est multiple d'une

même horloge de base. De même rien empêche de restreindre ce programme pour l'étude de politiques monoprocesseur. Le système d'équation initial Σ permettant notamment de vérifier l'ordonnançabilité en monoprocesseur est uniquement modifié sur l'une des conditions de validité (Eq. 5.1) :

$$\sum_{j \in J_k} w_{j,k} \leq 1, \forall k$$

Celle-ci exprime que la somme des poids doit être inférieure à la capacité du processeur sur chaque intervalle, ce qui est vrai pour toute stratégie optimale. Des tests complémentaires permettraient de comparer notre approche vis à vis d'autres ordonnanceurs optimaux monoprocesseur préemptifs (EDF, LLF, ...).

L'expérimentation s'est pour l'instant limitée à une simulation : l'implémentation de cette approche par un outil prototype et des mesures sur une architecture matérielle nous permettrait de mesurer les performances réelles par rapport aux autres algorithmes globaux à priorité fixe ou dynamique. Nous avons succinctement montré que notre approche ne se limitait pas à l'utilisation de tâches périodiques à échéances implicites. D'autres modèles de tâches mériteraient une étude plus approfondie par exemple des tâches non-cycliques [MNLM10, Bar10] ou synchrones (automates temporels [LDAV10]) avec la présence de branchements conditionnels : tous les traitements sont connus mais leur enchaînement peut varier. Ces modèles nécessiteraient la révision du modèle linéaire pour s'assurer de l'ordonnançabilité.

Enfin des modèles de tâches avec threads introduisent un nouveau degré de parallélisme au sein des tâches, permettant l'exécution de traitements d'une même tâche en parallèle. Dans notre travail, nous avons posé comme restriction qu'aucune tâche ne pouvait être exécutée en même temps par plusieurs processeurs. L'intérêt de ce concept est de réduire le temps de calcul global quitte à augmenter la charge de travail totale. Des politiques d'ordonnancement temps-réel de threads pourraient devenir de plus en plus intéressantes à la vue de la fabrication de MPSoC disposant de nombreux cœurs de calcul. Les méthodes basées sur ce type d'ordonnancement sont assez récentes. Colette et al. [CCG08] sont les premiers à avoir montrer la faisabilité d'un tel ensemble de tâches. Les algorithmes proposés multiplient les préemptions et migrations, ce qui les rend pratiquement très coûteux. Plus récemment, Lakshmanan et al. [LKR10] ont proposé de s'appuyer sur le modèle de programmation OpenMP, puis de transformer ce modèle de tâches avec fork-join en un modèle équivalent, dépourvu de ces structures par une méthode "d'étirement". La garantie de faisabilité se traduit néanmoins par une méthode d'augmentation de ressources. Ces méthodes proposées sont pour l'instant assez peu extensibles et efficaces. Il serait intéressant de voir en quoi notre travail pourrait s'appliquer à ce type d'ordonnancement.

Index

Bibliographie

[AAP08] Salvatore Carta Andrea Acquaviva, Andrea Alimonda and Michele Pittau. Assessing task migration impact on embedded soft real-time streaming multimedia applications. *EURASIP J. Embedded Syst.*, 2008 :1–15, 2008.

[AD98] Christophe Aussaguès and Vincent David. A method and a technique to model and ensure timeliness in safety critical real-time systems. *Engineering of Complex Computer Systems, IEEE International Conference on Operating systems principles (SOSP)*, 0 :0002, 1998.

[Ade02] A. Ademaj. Slightly-off-specification failures in the time-triggered architecture. In *Proceedings of the Seventh IEEE International High-Level Design Validation and Test Workshop*, pages 7–12, 2002.

[ADT08] Dalia Aoun, Anne-Marie Déplanche, and Yvon Trinquet. Pfair scheduling improvement to reduce interprocessor migrations. In Giorgio Buttazzo and Pascale Minet, editors, *16th International Conference on Real-Time and Network Systems (RTNS 2008)*, Rennes France, 2008. Isabelle Puaut.

[AFAL07] Luis Almeida, Sebastian Fischmeister, Madhukar Anand, and Insup Lee. A dynamic scheduling approach to designing flexible safety-critical systems. In *EMSOFT '07 : Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 67–74, New York, NY, USA, 2007. ACM.

[AfHOT06] Jim Alves-foss, W. Scott Harrison, Paul Oman, and Carol Taylor. The mils architecture for high-assurance embedded systems. *International Journal of Embedded Systems*, 2 :239–247, 2006.

[AK84] A. Avizienis and J. P. J. Kelly. Fault tolerance by design diversity : Concepts and experiments. *Computer*, 17 :67–80, August 1984.

[AK00] M. H. Azadmanesh and R. M. Kieckhafer. Exploiting omissive faults in synchronous approximate agreement. *IEEE Trans. Comput.*, 49 :1031–1042, October 2000.

[And03] Björn Andersson. *Static-priority scheduling on multiprocessors*. PhD thesis, 2003.

[ARI93] *ARINC Specification 659 : Backplane Data Bus*, 1993.

[ARI02] *ARINC 664, Aircraft Data Network, Part 1 : Systems Concepts and Overview*, 2002.

[AS99] Tarek F. Abdelzaher and Kang G. Shin. Combined task and message scheduling in distributed real-time systems. *IEEE Trans. Parallel Distrib. Syst.*, 10 :1179–1191, November 1999.

[AS00] James Anderson and Anand Srinivasan. Pfair scheduling : Beyond periodic task systems. In *In Proc. of the 7th International Conference on Real-Time Computing Systems and Applications*, pages 297–306, 2000.

[AT06] Bjorn Andersson and Eduardo Tovar. Multiprocessor scheduling with few preemptions. In *RTCSA '06 : Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 322–334, Washington, DC, USA, 2006. IEEE Computer Society.

[Ayd07] Hakan Aydin. Exact fault-sensitive feasibility analysis of real-time tasks. *IEEE Trans. Comput.*, 56 :1372–1386, October 2007.

[BaBW08] Eduardo Wenzel Brião, Daniel Barcelos, and Flávio Rech Wagner. Dynamic task allocation strategies in mpsoc for soft real-time applications. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '08, pages 1386–1389, New York, NY, USA, 2008. ACM.

[Bak05] Theodore P. Baker. Comparison of empirical success rates of global vs. partitioned fixed-priority and edf scheduling for hard real time. Technical report, Department of Computer Science, Florida State University, Tallahassee., 2005.

[Bar98] S. K. Baruah. A general model for recurring real-time tasks. In *RTSS '98 : Proceedings of the IEEE Real-Time Systems Symposium*, page 114, Washington, DC, USA, 1998. IEEE Computer Society.

[Bar03] Sanjoy K. Baruah. Dynamic and static priority scheduling of recurring real-time tasks. *Real-Time Syst.*, 24(1) :93–128, 2003.

[Bar10] Sanjoy Baruah. The non-cyclic recurring real-time task model. *Real-Time Systems Symposium, IEEE International*, 0 :173–182, 2010.

[BCA08] Björn B. Brandenburg, John M. Calandrino, and James H. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms : A case study. In *RTSS '08 : Proceedings of the 2008 Real-Time Systems Symposium*, pages 157–169, Washington, DC, USA, 2008. IEEE Computer Society.

[BCGM99] Sanjoy Baruah, Deji Chen, Sergey Gorinsky, and Aloysius Mok. Generalized multiframe tasks. *Real-Time Syst.*, 17(1) :5–22, 1999.

[BCPV93] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress : a notion of fairness in resource allocation. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, STOC '93, pages 345–354, New York, NY, USA, 1993. ACM.

[BCPV96] Sanjoy K. Baruah, N. K. Cohen, C. Greg Plaxton, and Donald A. Varvel. Proportionate progress : A notion of fairness in resource allocation. *Algorithmica*, 15(6) :600–625, 1996.

[Ber01] Joef Berwanger. Flexray–the communication system for advanced automotive control systems. In *SAE 2001 World Congress*, 2001.

[BFR71] Paul Bratley, Michael Florian, and Pierre Robillard. Scheduling with earliest start and due date constraints. *Naval Research Logistics Quarterly*, 18(4) :511–519, 1971.

[BGP95] Sanjoy K. Baruah, Johannes Gehrke, and C. Greg Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the 9th International Symposium on Parallel Processing*, IPPS '95, pages 280–288, 1995.

[BMR90] Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *In Proceedings of the 11th Real-Time Systems Symposium*, pages 182–190. IEEE Computer Society Press, 1990.

[Bor09] Étienne Borde. *Configuration et Reconfiguration des Systemes Temps-Reel Repartis Embarques Critiques.* PhD thesis, 2009.

[CCG08] Sébastien Collette, Liliana Cucu, and Joël Goossens. Integrating job parallelism in real-time scheduling theory. *Inf. Process. Lett.*, 106(5) :180–187, 2008.

[CFH+04] John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson, and Sanjoy Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models.* Chapman Hall/CRC, Boca, 2004.

[Che99] Pascal Chevochot. *Conception de systèmes temps-réel strict tolérants aux fautes.* PhD thesis, 1999.

[CMS82] X. Castillo, S. R. McConnel, and D. P. Siewiorek. Derivation and calibration of a transient error reliability model. *IEEE Trans. Comput.*, 31 :658–671, July 1982.

[Cof76] E. Coffman. Introduction to deterministic scheduling theory. *Computer and Job-Shop Scheduling Theory*, pages 1–50, 1976.

[CRJ06] Hyeonjoong Cho, Binoy Ravindran, and E. Douglas Jensen. An optimal real-time scheduling algorithm for multiprocessors. *Real-Time Systems Symposium, IEEE International*, 0 :101–110, 2006.

[Cru91] R.L. Cruz. A calculus for network delay. *IEEE Transactions on Information Theory*, 37(1) :114 –131, jan 1991.

[CSEF06] Hussein Charara, Jean-Luc Scharbarg, Jérôme Ermont, and Christian Fraboul. Methods for bounding end-to-end delays on an AFDX network. In *Euromicro Conference on Real-Time Systems (ECRTS), Dresden (Germany), 05/07/2006-07/07/2006*, pages 193–202, juillet 2006.

[CSM+05] Theofanis Constantinou, Yiannakis Sazeides, Pierre Michaud, Damien Fetis, and Andre Seznec. Performance implications of single thread migration on a chip multi-core. *SIGARCH Comput. Archit. News*, 33 :80–91, November 2005.

[DAL+04] V. David, C. Aussaguès, S. Louise, Ph, B. Ortolo, and C. Hessler. The OASIS Based Qualified Display System. In *Lecture Notes in Computer Science, 17th International Conf. on Computer Safety, Reliability and Security Fourth American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Controls and Human-Machine Interface Technologies (NPIC&HMIT 2004), Columbus, Ohio. September, 2004.*, September 2004.

[Den76] Peter J. Denning. Fault tolerant operating systems. *ACM Comput. Surv.*, 8 :359–389, December 1976.

[DGLS01] Catalin Dima, Alain Girault, Christophe Lavarenne, and Yves Sorel. Off-line real-time fault-tolerant scheduling. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0 :410, 2001.

[DM89] M.L. Dertouzos and A.K. Mok. Multiprocessor online scheduling of hard-real-time tasks. *Software Engineering, IEEE Transactions on*, 15(12) :1497–1506, Dec 1989.

[DM03] P. Dodd and L. Massengill. Basic mechanisms and modeling of single-event upset in digital microelectronics. *IEEE Trans. on Nuclear Science, Vol. 50, No.3*, June 2003.

[EB07] Paul Emberson and Iain Bate. Minimising task migration and priority changes in mode transitions. In *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*, pages 158–167, Washington, DC, USA, 2007. IEEE Computer Society.

[EB08] Paul Emberson and Iain Bate. Extending a task allocation algorithm for graceful degradation of real-time distributed embedded systems. In *Proceedings of the 2008 Real-Time Systems Symposium*, pages 270–279, Washington, DC, USA, 2008. IEEE Computer Society.

[eHmDFT07] Pierre emmanuel Hladik, Anne marie Déplanche, Sébastien Faucou, and Yvon Trinquet. Adequacy between autosar os specification and real-time scheduling theory. In *International Symposium on Industrial Embedded Systems*, pages 225–233, 2007.

[EJS+10] C. Engel, E. Jenn, P. H. Schmitt, R. Coutinho, and T. Schoofs. Enhanced dispatchability of aircraft using multi-static configurations. In *Embedded Real Time Software and Systems*, Toulouse, 2010.

[FKY08] Kenji Funaoka, Shinpei Kato, and Nobuyuki Yamasaki. Work-conserving optimal real-time scheduling on multiprocessors. In *Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, ECRTS '08, pages 13–22, Washington, DC, USA, 2008. IEEE Computer Society.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[GLS99] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors. In *Proceedings of the seventh international workshop on Hardware/software codesign*, CODES '99, pages 74–78, 1999.

[GM98] Juan A. Garay and Yoram Moses. Fully polynomial byzantine agreement for n > 3t processors in t + 1 rounds. *Siam Journal on Computing*, 27 :247–290, 1998.

[Gra66] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell Systems Technical Journal*, 45 :1563–1581, 1966.

[Gra69] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17 :416–429, 1969.

[GRE+01] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench : A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, 2001.

[GS78] Teofilo Gonzalez and Sartaj Sahni. Preemptive scheduling of uniform processor systems. *J. ACM*, 25(1) :92–101, 1978.

[HL92a] Ching-Chih Han and Kwei-Jay Lin. Scheduling real-time computations with separation constraints. *Inf. Process. Lett.*, 42(2) :61–66, 1992.

[HL92b] Kwang S. Hong and Joseph Y.-T. Leung. On-line scheduling of real-time tasks. *IEEE Trans. Comput.*, 41(10) :1326–1331, 1992.

[HLH99] J. Kim H. Lee and S. J. Hong. Evaluation of two load-balancing primary-backup process allocation schemes. *IEICE Transactions on Information and Systems*, 12, 1999.

[HLS77] Edward C. Horvath, Shui Lam, and Ravi Sethi. A level algorithm for preemptive scheduling. *J. ACM*, 24(1) :32–43, 1977.

[HS94] Chao-Ju Hou and K.G. Shin. Replication and allocation of task modules in distributed real-time systems. In *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, pages 26 –35, jun 1994.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

[IPEP05] Viacheslav Izosimov, Paul Pop, Petru Eles, and Zebo Peng. Design optimization of time-and cost-constrained fault-tolerant distributed embedded systems. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 2*, DATE '05, pages 864–869, 2005.

[JCD11] Mathieu Jan, Sylain Camier, and Vincent David. Scheduling safety-critical real-time bus accesses using time-constrained automata. In *19th International Conference on Real-Time and Network Systems (RTNS 2011)*, Nantes France, 2011.

[JDLP10] Mathieu Jan, Vincent David, Jimmy Lalande, and Maurice Pitel. Usage of the safety-oriented real-time oasis approach to build deterministic protection relays. In *SIES*, pages 128–135, 2010.

[JG99] Kevin Jeffay and Steve Goddard. A theory of rate-based execution. In *In Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS)*, pages 304–314. IEEE Computer Society Press, 1999.

[JKH05] Arshad Jhumka, Stephan Klaus, and Sorin A. Huss. A dependability-driven system-level design approach for embedded systems. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '05, pages 372–377, 2005.

[KAL04] Hamoudi KALLA. *Génération automatique de distributions/ordonnancements en temps-réel fiables et tolérants aux fautes*. PhD thesis, 2004.

[KDK+89] Hermann Kopetz, Andreas Damm, Christian Koza, Marco Mulazzani, Wolfgang Schwabl, Christoph Senft, and Ralph Zainlinger. Distributed fault-tolerant real-time systems : The mars approach. *IEEE Micro*, 9 :25–40, January 1989.

[KG94] Hermann Kopetz and Günter Grünsteidl. Ttp - a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27 :14–23, 1994.

[KK82] N. Karmarkar and R.M. Karp. The differencing method of set partitionning. Technical report, Berkeley, 1982.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume I : Fundamental Algorithms, 2nd Edition.* Addison-Wesley, 1973.

[KRSM09] Kedar M. Katre, Harini Ramaprasad, Abhik Sarkar, and Frank Mueller. Policies for migration of real-time tasks in embedded multi-core systems. In *In Work In Progress of the 20th IEEE Real-Time Systems Symposium (RTSS)*, 2009.

[KS86] C M Krishna and K G Shin. On scheduling tasks with a quick recovery from failure. *IEEE Trans. Comput.*, 35 :448–455, May 1986.

[KS97] Ashok Khemka and R. K. Shyamasundar. An optimal multiprocessor real-time scheduling algorithm. *Journal of Parallel and Distributed Computing*, 43 :37–45, 1997.

[KY09] Shinpei Kato and Nobuyuki Yamasaki. Semi-partitioned fixed-priority scheduling on multiprocessors. In *Proceedings of the 2009 15th IEEE Symposium on Real-Time and Embedded Technology and Applications*, RTAS '09, pages 23–32, 2009.

[Lat05] Elizabeth Latronico. Design time reliability analysis of distributed fault tolerance algorithms. In *DSN '05 : Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 486–495, Washington, DC, USA, 2005. IEEE Computer Society.

[LCC94] Tse Lee, Albert Mo Kim Cheng, and Kim Cheng. Multiprocessor scheduling of hard-real-time periodic tasks with task migration constraints. In *RTCSA*, 1994.

[LDAV10] M. Lemerre, V. David, C. Aussaguès, and G. Vidal-Naquet. An Introduction to Time-Constrained Automata. In *Proceedings of ICE 2010*, 2010.

[LDAVN08] Matthieu Lemerre, Vincent David, Christophe Aussaguès, and Guy Vidal-Naquet. Equivalence between schedule representations : Theory and applications. In *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 237–247, Washington, DC, USA, 2008. IEEE Computer Society.

[LFS+10] Greg Levin, Shelby Funk, Caitlin Sadowski, Ian Pye, and Scott Brandt. Dp-fair : A simple model for understanding optimal multiprocessor scheduling. In *Proceedings of the 2010 22nd Euromicro Conference on Real-Time Systems*, ECRTS '10, pages 3–13, Washington, DC, USA, 2010. IEEE Computer Society.

[LKP+10] Chanhee Lee, Hokeun Kim, Hae-woo Park, Sungchan Kim, Hyunok Oh, and Soonhoi Ha. A task remapping technique for reliable multi-core embedded systems. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES/ISSS '10, pages 307–316. ACM, 2010.

[LKR10] Karthik Lakshmanan, Shinpei Kato, and Ragunathan (Raj) Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, RTSS '10, pages 259–268, Washington, DC, USA, 2010. IEEE Computer Society.

[LKY+00] Yann-Hang Lee, Daeyoung Kim, Mohamed Younis, Jeff Zhou, and James McElroy. Resource scheduling in dependable integrated modular avionics. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*, DSN '00, pages 14–23, 2000.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1) :46–61, 1973.

[LP95] E.A. Lee and T.M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5) :773 –801, may 1995.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4 :382–401, July 1982.

[MC70] R. R. Muntz and E. G. Coffman. Preemptive scheduling of real-time tasks on multiprocessor systems. *J. ACM*, 17(2) :324–338, 1970.

[MC96] Aloysius K. Mok and Deji Chen. A general model for real-time tasks. Technical report, Austin, TX, USA, 1996.

[MC97] Aloysius K. Mok and Deji Chen. A multiframe model for real-time tasks. *IEEE Trans. Softw. Eng.*, 23(10) :635–645, 1997.

[MCDF09] Thomas Megel, Damien Chabrol, Vincent David, and Christian Fraboul. Dynamic Scheduling of Real-Time Tasks on Multicore Architectures. In *Colloque du GdR Soc/SiP*, orsay (91) France, 06 2009.

[McN59] R. McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 1959.

[MDP+00] Dejan S. Milojíčić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Comput. Surv.*, 32 :241–299, September 2000.

[MJDF11a] Thomas Megel, Mathieu Jan, Vincent David, and Christian Fraboul. Evaluation of task migration mechanisms for hard real-time distributed systems. In *RTNS'11 : Proceedings of the 19th International Conference on Real-Time and Network Systems*, 2011.

[MJDF11b] Thomas Megel, Mathieu Jan, Vincent David, and Christian Fraboul. Task migration mechanisms for hard real-time distributed systems. In *Proceedings WiP RTAS'11 : 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, 2011.

[MMT02] P.S. Miner, M. Malekpour, and W. Torres. A conceptual design for a reliable optical bus (robus). In *Digital Avionics Systems Conference, 2002. Proceedings. The 21st*, volume 2, pages 13D3–1 – 13D3–11 vol.2, 2002.

[MNLM10] Noel Tchidjo Moyo, Eric Nicollet, Frederic Lafaye, and Christophe Moy. On schedulability analysis of non-cyclic generalized multiframe tasks. *Euromicro Conference on Real-Time Systems (ECRTS '10)*, 0 :271–278, 2010.

[Mok83] A. K. Mok. Fundamental design problems of distributed systems for the hard-real-time environment. Technical report, Cambridge, MA, USA, 1983.

[Mos93] Daniel Mossé. *Design, Development, and Deployment of Fault-Tolerant Applications for Distributed Real-Time Systems*. PhD thesis, 1993.

[MSD10] Thomas Megel, Renaud Sirdey, and Vincent David. Minimizing task preemptions and migrations in multiprocessor optimal real-time schedules. In *Proceedings RTSS'10 : Proceedings of the 31st IEEE International Real-Time Systems Symposium*, page 37. IEEE Computer Society, 2010.

[MSM98] G. Manimaran, C. Siva, and Ram Murthy. A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis. *IEEE Trans. On Parallel and Distributed Systems*, 9 :312–319, 1998.

[NAG09] V. Nelis, B. Andersson, and J. Goossens. A synchronous transition protocol with periodicity for global scheduling of multimode real-time systems on multiprocessors. In *Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS) Work-in-progress session*, Washington D.C. USA, December 2009.

[NGA09] Vincent Nelis, Joel Goossens, and Bjorn Andersson. Two protocols for scheduling multi-mode real-time systems upon identical multiprocessor platforms. In *Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems*, pages 151–160, Washington, DC, USA, 2009. IEEE Computer Society.

[OS97] Yingfeng Oh and Sang H. Son. Scheduling real-time tasks for dependability. *Journal of Operational Research Society*, 48 :629–639, 1997.

[PABD+99] D. Powell, J. Arlat, L. Beus-Dukic, A. Bondavalli, P. Coppola, A. Fantechi, E. Jenn, C. Rabejac, and A. Wellings. Guards : a generic upgradable architecture for real-time dependable systems. *Parallel and Distributed Systems, IEEE Transactions on*, 10(6) :580 –599, jun 1999.

[Pad78] M. Padberg. Zero-one decision problems. Technical report, New York University, USA, 1978.

[PBD01] Sasikumar Punnekkat, Alan Burns, and Robert Davis. Analysis of checkpointing for real-time systems. *Real-Time Syst.*, 20 :83–102, January 2001.

[PBS+88] D. Powell, G. Bonn, D. Seaton, P. Verissimo, and F. Waeselynck. The delta-4 approach to dependability in open distributed computing systems. In *Fault-Tolerant Computing, 1988. FTCS-18, Digest of Papers., Eighteenth International Symposium on*, pages 246 –251, jun 1988.

[PBWB00] Stefan Poledna, Alan Burns, Andy Wellings, and Peter Barrett. Replica determinism and flexible scheduling in hard real-time dependable systems. *IEEE Trans. Comput.*, 49 :100–111, February 2000.

[PCGI04] Stefano Porcarelli, Marco Castaldi, Felicita Di Gi, and Paola Inverardi. A framework for reconfiguration-based fault-tolerance in distributed systems. In *in Distributed Systems, Architecting Dependable Systems II*. SpringerVerlag, 2004.

[PGH98] J. C. Palencia and M. González Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS '98, pages 26–, Washington, DC, USA, 1998. IEEE Computer Society.

[PH07] Michael Paulitsch and Brendan Hall. Insights into the sensitivity of the brain (braided ring availability integrity network)–on platform robustness in extended operation. *Dependable Systems and Networks, International Conference on*, 0 :154–163, 2007.

[Pol94] Stefan Poledna. Replica determinism in distributed real-time systems : a brief survey. *Real-Time Syst.*, 6 :289–316, May 1994.

[Pow95] David Powell. Failure mode assumptions and assumption coverage. Technical report, 1995.

[PSTW97] Cynthia A. Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation (extended abstract). In *STOC '97 : Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 140–149, New York, NY, USA, 1997. ACM.

[QHJ+00] Xiao Qin, Zongfen Han, Hai Jin, Liping Pang, and Shengli Li. Real-time fault-tolerant scheduling in heterogeneous distributed systems. In *in Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas*, pages 26–29, 2000.

[RC04] Jorge Real and Alfons Crespo. Mode change protocols for real-time systems : A survey and a new proposal. *Real-Time Syst.*, 26(2) :161–197, 2004.

[RGR08] Ahmed Rahni, Emmanuel Grolleau, and Michael Richard. Feasibility analysis of non-concrete real-time transactions with edf. In *16th International Conference on Real-Time and Network Systems RTNS2008 Rennes France*, Octobre 2008.

[Rou96] E. T. Roush. Fast dynamic process migration. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, pages 637–645, Washington, DC, USA, 1996. IEEE Computer Society.

[RR03] M.G.C. Resende and C.C. Ribeiro. Greedy randomized adaptive search procedures. *International Series in Operations Research & Management, Handbook of Metaheuristics*, 57, 2003.

[RS84] K. Ramamritham and J. A. Stankovic. Dynamic task scheduling in hard real-time distributed systems. *IEEE Softw.*, 1(3) :65–75, 1984.

[Rus01] John Rushby. A comparison of bus architectures for safety-critical embedded systems. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, sep 2001.

[Sah79] Sartaj Sahni. Preemptive Scheduling with Due Dates. *OPERATIONS RESEARCH*, 27(5) :925–934, 1979.

[SBEP11] Soheil Samii, Unmesh D. Bordoloi, Petru Eles, and Zebo Peng. Control-quality optimization of distributed embedded control systems with adaptive fault tolerance. *3rd Workshop on Adaptive and Reconfigurable Embedded Systems (APRES)*, 2011.

[SBS07] Marco Serafini, Andrea Bondavalli, and Neeraj Suri. Online diagnosis and recovery : On the choice and impact of tuning parameters. *IEEE Transactions on Dependable and Secure Computing*, 4, 2007.

[SBS+11] Marco Serafini, Peter Bokor, Neeraj Suri, Jonny Vinter, Astrit Ademaj, Wolfgang Brandstatter, Fulvio Tagliabo, and Jens Koch. Application-level diagnostic and membership protocols for generic time-triggered systems. *IEEE Trans. Dependable Secur. Comput.*, 8 :177–193, March 2011.

[SCKN07] Renaud Sirdey, Jacques Carlier, Hervé Kerivin, and Dritan Nace. On a resource-constrained scheduling problem with application to distributed systems reconfiguration. *European Journal of Operational Research*, 183(2) :546 – 563, 2007.

[SD08] Renaud Sirdey and Vincent David. Approches heuristiques des problèmes de partitionnement, placement et routage de réseaux de processus sur des architectures parallèles clusterisées. Technical report, 2008.

[She03a] Charles Shelton. *Graceful Degradation For Distributed Embedded Systems.* PhD thesis, 2003.

[She03b] Charles Preston Shelton. *Scalable graceful degradation for distributed embedded systems.* PhD thesis, Pittsburgh, PA, USA, 2003.

[Sir03] Renaud Sirdey. *Modèles et algorithmes pour la reconfiguration de systèmes répartis utilisés en téléphonie cellulaire.* PhD thesis, 2003.

[SJH+10] Neeraj Suri, Arshad Jhumka, Martin Hiller, András Pataricza, Shariful Islam, and Constantin Sírbu. A software integration approach for designing and assessing dependable embedded systems. *J. Syst. Softw.*, 83 :1780–1800, October 2010.

[SPG03] R. Sirdey, D. Plainfossé, and J.-P. Gauthier. A practical approach to combinatorial optimization problems encountered in the design of a high-availability distributed system. Evry, France, 2003.

[SR89] J. A. Stankovic and K. Ramamritham. The spring kernel : a new paradigm for real-time operating systems. *SIGOPS Oper. Syst. Rev.*, 23 :54–71, July 1989.

[SR04] John A. Stankovic and R. Rajkumar. Real-time operating systems. *Real-Time Syst.*, 28 :237–253, November 2004.

[Sri03] Anand Srinivasan. *Efficient and flexible fair scheduling of real-time tasks on multiprocessors.* PhD thesis, 2003. Director-Anderson, James H.

[SS75] J.H. Saltzer and M.D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9) :1278 – 1308, sept. 1975.

[TBEP10] Bogdan Tanasa, Unmesh D. Bordoloi, Petru Eles, and Zebo Peng. Scheduling for fault-tolerant communication on the static segment of flexray. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, RTSS '10, pages 385–394, 2010.

[TC94] Ken Tindell and John Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocess. Microprogram.*, 40 :117–134, April 1994.

[TCN00] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proceedings International Symposium on Circuits and Systems*, volume 4, 2000.

[TLC85] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the v-system. *SIGOPS Oper. Syst. Rev.*, 19 :2–12, December 1985.

[TS94] Sandra R. Thuel and Jay Strosnider. Scheduling fault recovery operations in time-critical systems. 1994.

[TTT] TTTech. *Advanced Control Systems for Boeing 787 Dreamliner Hamilton Sundstrand's TTP-Based Communication Platform.*

[Vil88] Alain Villemeur. *Sûreté de fonctionnement des systèmes industriels.* Ed. Eyrolles, 1988.

[WCL+07] Hsin-Wen Wei, Yi-Hsiung Chao, Shun-Shii Lin, Kwei-Jay Lin, and Wei-Kuan Shih. Current results on edzl scheduling for multiprocessor real-time systems. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '07, pages 120–130, Washington, DC, USA, 2007. IEEE Computer Society.

[XP93] J. Xu and D. L. Parnas. On satisfying timing constraints in hard-real-time systems. *IEEE Trans. Softw. Eng.*, 19(1) :70–84, 1993.

[ZMC03] Dakai Zhu, Rami Melhem, and Bruce R. Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multiprocessor real-time systems. *IEEE Trans. Parallel Distrib. Syst.*, 14(7) :686–700, 2003.